CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r. 82/08)

Canada

0-315-08953-9

National Library of Canada
Canadian Theses Division
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Division des thèses canadiennes

65

56870

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

EDWARD SACHARUK

Date of Birth — Date de naissance
JAN 10, 1948

Country of Birth — Lieu de naissance
England

Permanent Address — Résidence fixe
6804 - 106 Avenue, Edmonton, Alberta T6A 1G9

Title of Thesis — Titre de la thèse

Secure Personal Communications and Authentication

University — Université
University of Alberta

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée
Master of Science in Computing Science

Year this degree conferred — Année d'obtention de ce grade
1982

Name of Supervisor — Nom du directeur de thèse
L.W. Jackson

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date
Dec. 7 1981

Signature
Ed Sacharuk

NL-91 (4/77)

THE UNIVERSITY OF ALBERTA

SECURE PERSONAL COMMUNICATIONS AND AUTHENTICATION

by

© EDWARD SACHARUK

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON ALBERTA CANADA

SPRING 1982

THE UNIVERSITY OF ALBERTA

RELEASE FORM

Name of Author: EDWARD SACHARUK

Thesis Title: SECURE PERSONAL COMMUNICATIONS AND AUTHENTICATION

Degree: MASTER OF SCIENCE

Year Granted: 1982

Permission is hereby granted to the UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly, or scientific purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's permission.

Ed Sacharuk

Permanent Address:

6804-106 Avenue
Edmonton, Alberta
Canada T6A 1G9

Date Dec. 4, 1981

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "SECURE PERSONAL COMMUNICATIONS AND AUTHENTICATION", submitted by Edward Sacharuk in partial fulfillment of the requirements for the degree of Master of Science.

.....L. W. Jackson.....

L.W. Jackson (supervisor)

.....J. J. Kates.....

.....S Cals.....

Date...Nov. 30., 1981....

Abstract

An inexpensive and portable secure communications and authentication network might find some application but to date the designs proposed have been expensive and severely restricted in their locatability. Proposed networks, implementing either private- or public-key cryptosystems, require a large securely located central computer for key generation, initiation of communications channels, and authentication.

This thesis examines the problem of designing a secure network that uses only small computers for the communications devices and the central mechanism. It is shown that such a network can be constructed if a public-key cryptosystem is implemented with keypair generation distributed to the communications devices themselves. Besides reducing the workload of the central computer, distributed keypair generation is shown to have the side-effect that a protocol can be designed that makes it very difficult for an intruder to undetectably use a lost or stolen secret key.

The thesis also examines the feasibility of implementing a public-key cryptosystem, in its entirety, on microcomputers. It is shown that one public-key algorithm, the RSA cryptosystem, has potential for such application but that in a straightforward implementation encryption speed and keypair generation speed will be too slow for a useful

network to be constructed. A method for increasing these speeds is presented that involves the use of a new algorithm for finding the remainder of a division; the improved performance is shown to be sufficient to make practical the construction of a microprocessor-based secure communications and authentication network suitable for interpersonal applications.

Acknowledgements

Theses are not written in a vacuum. This one is to a large degree a byproduct of the help given me in many ways by my fellow students and the faculty of the Department of Computing Science during the past three years. It has been a wonderful and exciting three years. Thank you.

This thesis would not have been conceived without a particularly fruitful discussion with Tony Olekshy in the summer of 1980. It would not have reached its present form without the many helpful, and always correct, criticisms and suggestions by my supervisor, Professor Wayne Jackson.

Dedication

Hwee Kin: no words

Jan: no numbers

Table of Contents

Introduction 1

Chapter One - A Cryptography Primer 7

1.1 Introduction 7

1.2 Basics 10

1.3 DES 14

1.4 Public-Key Cryptography 16

1.5 The RSA Cryptosystem 20

1.6 Electronic Signatures with the RSA Cryptosystem 24

Chapter Two - An Improved Secure Communications Network 26

2.1 Introduction 26

2.2 Some Secure Communications Network Concepts 28

2.3 Historical Perspective 33

2.4 A Secure Communications and Authentication Network 37

2.4.1 A Trivial Public-Key SCD 37

2.4.2 An Improved KDC and Protocol 39

2.4.3 Further Considerations 46

2.5 Applications 50

2.6 Summary and Conclusions 51

Chapter Three - RSACRYPT: An Implementation 53

3.1 Introduction 53

3.2 The Program 54

3.3 Difficulties 58

3.3.1 Quasi-Random Number Generation 58

3.3.2 Prime Number Generation 60

3.4 Analysis of Results 63

3.4.1 RSACRYPT Results 63

3.4.2 Michelman's Results 66

3.4.3 Projected Encryption Speed on the MC68000 67

3.4.4 Projected Key Generation Speed on the MC68000 68

3.4.5 Projected Rates With Improved Algorithms 70

3.4.6 Projected Network Rates 71

3.5 Conclusions 72

Chapter Four - Towards Faster Modular Exponentiation 73

4.1 Introduction 73

4.2 The Modular Exponentiation Problem 74

4.3 Approaches that are Infeasible In Practice 76

4.4 Approaches that Improve Timing 83

4.4.1 Multiplication using the Karatsuba Algorithm 83

4.4.2 The Preconditioned Fast Remainder Algorithm 87

4.5 Conclusion 95

Summary and Conclusions 98

References 100

Appendix 1 - An Example of DES 104

List of Figures

2.1 Private-Key SCN's 31

2.2 The Proposed KDC 40

3.1 Operation of RSACRYPT 55

A.1 The e-table 106

A.2 The s-tables 107

A.3 Encryption with DES 110

A.4 Decryption with DES 112

List of Tables

1.1 Time to Factor n 23

3.1 Encryption Speed of RSACRYPT 64

3.2 Key Generation Time of RSACRYPT 65

3.3 Michelman's Results 66

3.4 Projected Encryption Speeds on the MC68000 67

List of Algorithms

4.1a. ms 93

4.1b. mhote 93

4.1c. collapse 94

4.1d. pfra 94

Introduction

Efforts have been made throughout history to build a secure communications network, but only recently has it become possible to build a network that is inexpensive and portable as well as secure. Further, it is now possible to provide network users with convenient and unforgeable means of message authentication or 'electronic signatures'. An inexpensive secure communications and authentication network, if constructed, would have numerous applications.

This thesis examines the problem of designing a secure personal communications and authentication network that implements a 'public-key' cryptosystem on microcomputers in software. Individual microcomputers would be linked together as a network having at its center a central trusted mechanism or 'key distribution center' implemented in software on a small computer or microcomputer. Since the rates of encipherment, decipherment, and message signing are expected to be low using small computers the network will be useful only for *personal* communications and authentication and not in areas involving the rapid transfer of large quantities of information.

The problem of designing a secure microprocessor-based network is approached on four levels in Chapters One to Four, summarized as follows:

Chapter One. To attain the desired goals it is first necessary to understand something about cryptography in general and public-key cryptosystems (there is more than one) in particular. This knowledge will aid in deciding which public-key algorithm to implement, an important consideration since public-key algorithms are not all of equivalent quality.

Chapter One is a review of cryptography in which particular emphasis is placed on DES (the 'Data Encryption Standard'), a conventional cryptosystem, and on the Rivest-Shamir-Adleman (RSA) public-key cryptosystem. We believe these two algorithms to be the highest-quality representatives of their classes; both find application in the software designed.

Chapter Two. Designing a secure communications network involves more than simply implementing an algorithm on computers: a network must be designed also, and this requires the design of a central mechanism for authentication, key distribution, and initiation of communications channels, as well as a protocol for its use. These considerations, among others, are studied in Chapter Two; a key distribution center and a protocol are designed that in some ways appear to be novel and which make an inexpensive secure network possible. The designs involve distributing the keypair generation process to the network nodes, which is done for three reasons:

(1) Distribution of keypair generation allows keys to be changed more frequently than in a network with centralized generation, making cryptanalysis harder.

(2) The central mechanism can be simplified thereby making it more trustworthy. Also, it will be easier to implement a simple central mechanism on a small computer.

(3) Detection of security breaches is more rapid than in previous designs, a consideration that is of particular importance in preventing the forgery of signatures.

The design of an inexpensive secure communications network is interesting from a purely theoretical standpoint, but there is also a practical side. This chapter concludes with a list of some possible applications of the proposed design.

Chapter Three. This chapter is a discussion of an implementation of the RSA cryptosystem that we constructed at the University of Alberta, the difficulties faced and overcome in implementation, and an analysis of the results obtained by running the program.

One important difficulty that was faced arises from the requisite complexity of the program: all public-key cryptosystems are complex so there are problems in implementing one on a large computer, let alone a microcomputer. An implementation must be trustworthy: the

user must have confidence in his software and it should work correctly every time. Further, some difficulties occur because of the necessity of generating large quasi-random numbers and prime numbers. Despite the requirement of trustworthiness and the difficulties to be overcome we show that with appropriate structuring the software can be implemented relatively easily.

It is also shown in Chapter Three that with a simple 'trick' it is possible to generate keypairs 5 times faster than by using a straightforward approach, and that it is reasonable for the RSA cryptosystem to be implemented in software on a powerful microcomputer such as the MC68000. Keypairs will be generated rapidly enough, using the MC68000, to implement the network designed in Chapter Two and encryption speed will be fast enough to satisfy a better than average typist.

Although the software described implements the RSA cryptosystem, which may be found to be insecure or superceded by better, faster, algorithms, our program should be of some help in implementing another public-key algorithm; the design principles and much of the software will be easily adapted to construction of future software.

Chapter Four. Since public-key algorithms encipher information slowly, the major difficulty with a software solution to the proposed network is the expected low throughput on communications channels if microcomputers are

used. Each of the known public-key cryptosystems require that a time-consuming computation be repeatedly done to encipher or decipher information. The RSA algorithm uses a technique called the 'finite exponential' or 'modular exponentiation' to encrypt messages; the best algorithm known for modular exponentiation (the method of repeated squaring and multiplication) has a time complexity of $O(n^3)$ if the standard algorithms for multiplication and division are used.' [2]

Chapter Four is a discussion of approaches taken towards computing the finite exponential as quickly as possible in practice, with the goal of enabling network users to encipher information at typing speed, using a high-quality cryptographic key, on a microcomputer. It is shown that it is possible to carry out modular exponentiation almost three times faster in practice than with use of the standard algorithms. Part of this gain in speed is achieved through the use of an algorithm for finding the remainder of a division that is, as far as we know, original.

' The standard algorithms are the ones commonly used for manual multiplication and division. They are detailed in algorithms 'M' and 'D' by Knuth[29].

[2] Throughout the thesis we use 'n' in two different ways: in the 'O-notation', as in $O(n^3)$, to indicate the time complexity of algorithms, and as the modulus used in the RSA public-key cryptosystem. The usage will be clear from the context.

Chapter Four also includes a description of a parallel algorithm for multiplication that can be easily implemented in hardware, to provide a very large rate of encryption.

Despite the relatively low encryption speeds that are possible in software, even with improved algorithms, it will be seen in this thesis that it is not entirely correct to say:

> "A necessary component of such a public-key system implementation ... is a hardware device for rapid modular exponentiation."[51]

In fact, an entire spectrum of networks implementing the RSA cryptosystem can be built, with varying rates of operation depending on whether microprocessors, minicomputers, mainframes, or special hardware devices are used. Each type of implementation may find application, depending upon the requirements of network users and their resources.

Chapter One

A Cryptography Primer

1.1 Introduction

Cryptography is a collection of methods for concealing information. A *logical channel*[40] is created whenever two or more parties, separated in time or space or both, communicate by enciphering data using a cryptosystem. Parties that communicate using a cryptosystem can have a *secure* conversation but not a *secret* one, since a third party can determine that a conversation is taking place but not the contents of the message.[1] Depending on the method used (the *cryptosystem*) and its means of implementation the rate of communication between parties may be large or small; that is, the logical channel may be fast or slow.

Since modern communications traffic requires large amounts of confidential information to be handled daily, the utility of a cryptosystem is partly measured in terms of the *channel capacity*, which is the maximum amount of information that can be enciphered and transmitted in a unit of time on a particular logical channel. Research continues for fast

---

[1] Secure information traffic, since it is detectable, is subject to *traffic analysis* which is discussed by Chaum[6]. Secret information traffic requires the use of *steganography*[26] to conceal the existence of private communications channels.

means of carrying out encryption[24,51] as well as for cryptosystems that are highly (possibly even provably) secure.

The ideals of large channel capacity and high *cryptosecurity* seem to be somewhat incompatible, however. The conventional, symmetric[46] or *private-key cryptosystems* (systems in which both communicating parties use the same cryptographic key) can be implemented to have large channel capacities, but recently doubts have been raised about the long-term cryptosecurity of the best of these, DES. Some of the new asymmetric[46] or *public-key cryptosystems* (systems in which the communicating parties use inverse keys to encipher and decipher information) seem to have mathematical reasons for believing them secure but they encipher information relatively slowly.[1]

---

[1] The wide disparity between the channel capacities of private- and public-key implementations has been noted in the literature:

1. Chip implementations of DES, and at least one software implementation, operate at 10^6 to 10^7 bits per second (bps) (Williams and Hindin[50]).

2. The RSA public-key algorithm has been implemented in software on a large computer to encipher at approximately 500 bps (Michelman[36]) and a 5000 bps hardware version may eventually be constructed (Denning[9]).

Simmons[46] has observed that public-key algorithms encipher at rates not greater than $C^{1/2}$ where C is the encryption speed of private-key implementations.

Despite the small channel capacities possible with public-key algorithms, however, they have some advantages. One of the two inverse keys (the *public* key) can be transmitted openly over an insecure channel, to be used for encipherment of messages to the key owner; the owner deciphers using his *secret* key. The use of inverse keys allows unforgeable *electronic signatures* to be easily implemented.

Kahn[26] and other researchers[7,31] have comprehensively surveyed private-key cryptosystems and their evolution. Extensive treatments of the theory[15,21,27,45] and implementation[32] of private-key systems has been published, as well as proposals for standardization[38]. Much discussion of public-key cryptography has been published since their conception by Merkle in the late 1970's, including surveys[8,12,17,22,23,31,46], theory and presentations of new systems[34,35,42], a discussion of implementation[36], and a discussion of some directions that could be taken in the future in designing improved public-key algorithms[18]. The merits of DES and public-key systems have been argued by their proponents[48].

The remainder of this chapter briefly reviews the state of knowledge in cryptography with particular emphasis placed on concepts and algorithms referred to throughout the thesis.

1.2 Basics

Perhaps the simplest private-key cryptosystem is the *Caesar cipher*, attributed to Julius Caesar. In this system the alphabet is written twice, in two rows, the second row below and shifted with respect to the first. To encrypt a message, characters in the message are looked up in the first row and the corresponding characters in the second row are written out. The Caesar cipher is the simplest *shift cipher*.

The shift cipher is the basis of *polyalphabetic ciphers* that, in essence, use a different shift for each character of a message. These have led to the *one-time pad*, a provably unbreakable cryptosystem that requires the use of an encryption key as long as the message to be encrypted; although the one-time pad is an *unconditionally secure* cryptosystem, its use is *computationally infeasible* for most applications[12].

Polyalphabetic ciphers are members of a broader class of cryptosystems known as *substitution ciphers* that replace symbols in the plaintext message with other symbols, with no change in order. Ciphers that do permute the order of symbols are called *transposition ciphers*.

Product ciphers are combinations of substitution and transposition ciphers such that if S is a substitution cipher with $C_1 \leftarrow S(M)$ where M is the message to be encrypted, and if T is a transposition cipher such that

$C_2 \leftarrow T(M)$ then P, the product cipher of S and T, is defined by

$$C_3 \leftarrow P(M) = S(T(M)) = T(S(M))$$

where C_1, C_2, and C_3 are ciphertext. Product ciphers generally involve numerous cycles of substitution and transposition.

All cryptosystems require at least one *key* to control the encryption process. For all cryptosystems it is true that:

$$C \leftarrow E(M,K_1)$$

$$M \leftarrow D(C,K_2)$$

where C and M are the ciphertext and plaintext respectively, E and D are the encryption and decryption functions respectively, and K_1 and K_2 are the encryption and decryption keys, respectively.

In private-key systems K_1 and K_2 are the same and control the substitutions, transpositions, or both, that take place during encryption or decryption. Both the sender and receiver must therefore have access to the same key, which may be transmitted from one to the other by private courier, for example. The functions E and D are not the same, but act as inverses in that a message block that is encrypted using either function and a particular key is restored to its original state when decrypted using the other function and the same key.

On the other hand, in public-key cryptosystems E and D are precisely equivalent but the keys are different, acting as inverses of each other. One of the keys can be revealed with no loss of security but with the possibility that the key owner may receive unwanted messages from unknown parties.

Shannon[45] laid the foundation for mathematical analysis of cryptography, particularly private-key cryptography, by introducing the concepts of *confusion* and *diffusion*. Confusion is an easily quantifiable concept that is provided by all substitution ciphers to varying degrees. Diffusion is less easily quantified since it deals with disguising the statistical characteristics (*statistics*) possessed by all spoken languages; statistics are related in a complex way to the redundancy inherent in spoken languages and a precise measure of redundancy is difficult to formulate. Statistical information can be obtained from analysis of the frequency of occurrence of single letters or groups of letters (*digram*, *trigram*, and *n-gram* statistics). Transposition ciphers attempt to conceal statistics by diffusing them into a featureless 'background noise'.

Cryptanalysis involves the use of n-gram statistics or other *side information* to decipher a cryptogram without knowledge of the key used. Side information can come from sources other than statistical analysis. For example, *probable word analysis* involves searching for expected words in a cryptogram; a corporate cryptogram might be expected to

use the words "profit" and "loss" for instance.

The best source of side information is the plaintext message itself. The strongest possible attack against a cryptosystem, known as the *chosen plaintext attack*, occurs when a cryptanalyst can encrypt a message of his choosing. Weaker methods of attack are the *known plaintext* attack (in which the cryptanalyst is given a quantity of ciphertext and its corresponding plaintext) and the *ciphertext only* attack. These attacks are used for the testing of private-key systems, with public-key cryptosystems depending on mathematical reasoning to show them secure. Regardless of the attack used the quantity of enciphered messages is important; modern cryptosystems cannot be cryptanalysed with only a small amount of plaintext and ciphertext.

All cryptosystems can be characterized as either *block* or *stream* ciphers. Stream ciphers encipher characters individually as they are passed into the implementation whereas block ciphers encrypt entire blocks of characters at a time, causing each bit in the enciphered block to be interrelated with all other bits. Modern private-key block ciphers work with blocks of 4, 8, or 16 characters, while public-key systems generally use variable-length blocks whose length depends upon the key length, also a variable. Notice that stream ciphers are degenerate versions of block ciphers.

Finally, it is possible to use *feedback* to combine the encrypted output with the input in some fashion (usually the 'exclusive or'). Feedback is called *block chaining* when used in conjunction with a block cipher and causes an encrypted block to be related to all previously encrypted blocks as well as the key. More than one cycle of chaining of the blocks of a message creates a complex interrelationship among all bits in a cryptogram.

1.3 DES

Recognizing the need for an industry standard for cryptography, the American National Bureau of Standards (NBS) conducted a competition among private companies to arrive at such a standard. IBM, apparently in consultation with the National Security Agency, won the competition with a derivative of a block cipher called Lucifer (developed by Tuchman) that used a key of 128 bits on blocks of 16 characters. Lucifer's derivative was tested by the NBS, found satisfactory, and promulgated as DES. DES uses a 56-bit key that is artificially expanded to be 64 bits in length before use for encryption; it is a block, product, private-key cipher.

For an implementation to be advertised as conforming to the standard the NBS requires that it be in hardware, presumably for testing purposes. Since DES uses some tables, the s-tables and e-table, for *non-linear*

substitution (in which a group of bits in a message is replaced by fewer, or more, bits) the tables are unmodifiable in a standard implementation.

Largely through the efforts of Hellman of Stanford University, a controversy has arisen surrounding DES calling into question its ability to adequately safeguard information, particularly in the long term. At least three areas in which DES may be suspect have been noted:

(1) *The Short Key.* Hellman feels that it is possible to build a $20,000,000 parallel computer, using available technology, that would allow its owner to cryptanalyse DES-enciphered messages quickly. Consequently, he believes that the NBS should have stayed with the 128-bit key used in Lucifer.

(2) *The Tables.* It is conceivable that a 'trapdoor' was built into the fixed tables to allow the NSA to easily decipher encrypted data without knowledge of the key used. The principles underlying the construction of the tables have never been revealed by IBM.

(3) *The Testing.* Although DES was attacked for many man-hours by the NBS the weaker known-plaintext attack was used.

(For further information consult Lempel[31] and Sugarman[48] who discuss the *DES controversy* fully.)

In spite of its possible flaws, DES is the best publicly available private-key cryptosystem. Appendix 1 provides a brief example of the operation of a simplified version of DES that may aid in understanding the algorithm, at least on an intuitive level. The example is derived from explanations by Lempel[31] and Hindin[24].

1.4 Public-Key Cryptography

Public-key cryptography is founded upon the use of outstandingly difficult mathematical problems, which are inverted in some sense and used as bases for cryptosystems. Whether the cryptosystems themselves are as difficult to solve as their underlying base problems is still unclear in most cases', but there are reasons for believing that at least one of the algorithms (the RSA system) has the same complexity as its base problem.

The paradigm for public-key cryptography can be expressed as

$$C \leftarrow F(M,K_1)$$

$$M \leftarrow F(C,K_2)$$

where M is the plaintext, C is the ciphertext, K_1 is the encryption key and K_2 is the decryption key. The paradigm

---

' For example, Lempel[31] outlines a public-key cryptosystem that is based on an NP-complete problem, yet is relatively easy to cryptanalyse.

requires the use of two difficult problems for its realization. To show why this is so, we define a *one-way function* and a *trapdoor one-way function*:

(1) A function f is said to be *one-way* if it is invertible and easy to compute, but for almost all x it is computationally infeasible to solve y=f(x) for x, given y. That is, f is a one-way function if its inverse is very difficult to compute.

(2) It is sometimes possible to arrange that the inverse of a one-way function is easy to compute given some additional information. In this case there is a *trapdoor* between f and its inverse and f is a trapdoor one-way function.

From these definitions it can be seen that K_1 and K_2 in the public-key paradigm must be related by a trapdoor one-way function so that it is difficult to compute one from the other. Additionally, knowledge of F and either M or C without the corresponding key must be insufficient to compute C or M, respectively.

The literature describes six public-key schemes, with some reference made to unpublished proprietary schemes. The Rivest-Shamir-Adleman (RSA) cryptosystem, seemingly the best of the six, is discussed later in this chapter. The other five public-key cryptosystems are briefly summarized below, with emphasis on their flaws:

(1) *Merkle Cryptosystem.* This algorithm (devised by Merkle[34]) requires the 'enpuzzlement' of n puzzles, each of which requires O(n) time to solve, by the transmitter of a message. The receiver solves one of the puzzles and sends its number and solution. The transmitter knows the solutions to his own puzzles and so is able to use the one solved by the receiver to derive a key for encryption of a message. An eavesdropper must do $O(n^2)$ work to decipher the message because he must first solve n/2 puzzles, on average.

The Merkle cryptosystem is unuseable because it requires only that a cryptanalyst do $O(n^2)$ work while the transmitter do O(n), which is too low a ratio of work factors. Hellman[22] states that it might be used in the future with fiberoptic technology, but this seems doubtful. He also points out that the method is the "simplest and least likely to yield to cryptanalysis".

(2) *Diffie-Hellman Cryptosystem.* This scheme employs the use of the 'discrete exponential' (i.e., modular exponentiation) for key exchange. It seems an excellent method but according to Hellman[22] it "needs study". Signatures don't seem possible because each user has the same key after the exchange of some numbers; that is, the method reduces to a sort of private-key method.

(3) *Merkle-Hellman Cryptosystem.* 'Trapdoor Knapsacks' form the basis of this cryptosystem devised by Merkle and Hellman[35]. Lempel[31] notes two flaws:

- Simple digital signatures seem impossible.
- Although the general knapsack problem is NP-complete, there is no proof that the trapdoor knapsack problem is also NP-complete.'

(4) *Graham-Shamir Cryptosystem.* This algorithm has not yet been published and the only available account (by Lempel[31]) is sketchy. It involves a variation on the Merkle-Hellman trapdoor knapsack concept. At least one very large table seems to be required per user and signatures seem as difficult to implement as in the Merkle-Hellman method.

(5) *McEliece Cryptosystem.* This algorithm is based on the 'general decoding problem for error-detecting codes'[31] which has been shown to be NP-complete. That is, it uses algebraic coding theory and 'scrambled Goppa error-detecting codes'[22] to define keys. At present no easily implemented signature scheme is possible[31] and a large space requirement, 500 kilobits, is needed for a 'generator matrix'[22].

' Problems lying in the class called NP are believed to have time complexities that are not expressible as a polynomial, if implemented on a deterministic computer.

1.5 The RSA Cryptosystem

The RSA cryptosystem is distinguished from the other public-key algorithms in two important ways. First, only the RSA method has an easily-implemented message-dependent signature facility. Although on the surface all public-key cryptosystems seem to be equivalent, in practice the other public-key systems do not permit the consecutive use of two different keys on the same message, which must be possible to sign messages without special measures being needed (such as authenticating messages by sending them to a central computer, for example).

Second, and more important, the RSA method has withstood concerted attack for some time. Simmons and Norris[47] had a possible attack on the system refuted by Rivest[41]. Lempel[31] cites an unpublished proof by Rabin that places the RSA method in a "safe position as long as factorization remains hard". Factorization has not been proved hard however, so it would be unwise to place complete trust in the system until a proof is found. Cabay[5] has observed that some probabilistic attacks presently under development may eventually allow rapid factorization in many cases, thereby permitting the easy solution of some cryptograms. By then it may be hoped that another good cryptosystem will be found.

The RSA cryptosystem has been clearly described[42]. What follows is a summary.

RSA keypairs are triples (e,d,n) where the public key is the pair (e,n) and the secret key is (d,n). d is a large prime number, e is the *multiplicative inverse* of d, and n is the product of two large primes p and q. Therefore, in terms of the public-key paradigm the RSA method can be expressed as:

$$C \leftarrow F(M,(e,n))$$

$$M \leftarrow F(C,(d,n)).$$

Since 'F' in the RSA method is *modular exponentiation*, encryption and decryption can be expressed more precisely by:'

$$C \leftarrow M\uparrow e \pmod n$$

$$M \leftarrow C\uparrow d \pmod n$$

where (e,d,n) is the receiver's keypair.

Modular exponentiation provides a trapdoor one-way function since it is easy to obtain C from M, using (e,n), but the receiver must have the additional information (d,n) to obtain M from C.

As shown earlier, public and secret keys in all cryptosystems must also be related by a trapdoor one-way function. In the RSA algorithm, e and d are multiplicative inverses modulo the *Euler totient* of n. The Euler totient,

---

' Throughout the thesis '↑' indicates exponentiation.

t(n), is the quantity of numbers less than n and relatively prime to n and in its simplest form is (p-1)×(q-1). Without knowledge of t(n) it is an enormously difficult problem to obtain d from e because this requires factoring n to obtain p and q.'

In other words, one of the two trapdoor functions in the RSA cryptosystem utilizes the difficulty of inverting the computation of modular exponentiation without knowing the secret key d, and the other function utilizes the difficulty of inverting the computation of the multiplicative inverse, to obtain d, without knowing the factors of n.

Rivest, Shamir, and Adleman[42] report that the best algorithm for factoring n is by Schroeppel (unpublished) which takes the times listed in Table 1.1 (duplicated from Rivest, et al.[42]). Rivest, et al., recommend that n be 200 decimal digits, although a smaller n will still be very difficult to factor.

Computing an RSA keypair is straightforward:

(1) Obtain 3 quasi-random numbers d', p', and q'. Find the primes d, p, and q greater than or equal to d', p', and q'. d must lie in the range

---

' In fact, the common factors of p-1 and q-1 should be extracted from t(n); this is easy to do using the Extended Euclid's Algorithm (see Knuth[29]). Extraction of common factors reduces the size of e that is derived from d using t(n). Reduction of the size of e speeds up encryption and is therefore important in practice.

n length (decimal digits)	Number of Operations to Factor n	Time to Factor n (@1 operation/ microsecond)
50	1.4×10^{10}	3.9 hours
75	9.0×10^{12}	104 days
100	2.3×10^{15}	74 years
200	1.2×10^{23}	3.8×10^{9} years
300	1.5×10^{29}	4.9×10^{15} years
500	1.3×10^{39}	4.2×10^{25} years

Table 1.1. Time to Factor n
(from Rivest, Shamir, and Adleman)

$$\max\{p,q\} < d < p\times q.$$

Compute $n=p\times q$. (d,n) is the desired secret key.[1]

(2) Compute $(p-1)\times(q-1)$ and extract the common factors of $p-1$ and $q-1$ to reduce the size of e computed later. This is the Euler totient of n, t(n).

(3) Compute e from $e\times d \equiv 1$ (modulo t(n)) using Euclid's algorithm.[2] [3]

Although it seems that the RSA method can be broken only by factorization there is another point of attack: the required quasi-random number generator. If the quasi-random numbers generated are insufficiently random then it might be

---

[1] More precisely, d need only be relatively prime to $(p-1)\times(q-1)$.

[2] '$\equiv$' indicates congruency throughout the thesis.

[3] It is the fact that $e \times d \equiv 1$ (modulo t(n)) and not $e \times d \equiv 1$ (modulo n) that makes it difficult to decipher a cryptogram; otherwise the secret key d could be easily derived from $d = n/e$.

possible, given e, n, the random number generator's mode of operation, and a clever cryptanalyst, to somehow compute d.

1.6 Electronic Signatures with the RSA Cryptosystem

Electronic message signing is described in detail in [8,41]. The central ideas underlying the concept are briefly indicated here, with reference to the RSA cryptosystem.

Let S be some function of the message M and, perhaps, the keypair-owner's name and other desired information.[1] For the owner A to send a signed message to B he computes

$$CS \leftarrow S \uparrow d_1 \pmod{n_1}$$

where CS is the encrypted signature and (d_1, n_1) is A's secret key.

Let M+CS be the message concatenated with the encrypted signature and (e_2, n_2) be B's publicly available key. A sends B the cryptogram

$$C \leftarrow (M+CS) \uparrow e_2 \pmod{n_2}.$$

---

[1] The "function of the message M" referred to can be either the entire message or, if desired, a hash function of M, h(M), which allows a 'compressed signature'[8]. It must be remembered that more than one message will hash to the same signature so that, with a knowledge of the hash function, an unscrupulous user of the cryptosystem could make it appear that a second user signed a document that the second user did not sign. Throughout this thesis it is assumed that the entire message M is included in the signature.

Receiver B obtains M+CS by computing

$$M+CS \leftarrow C+d_2 \pmod{n_2}.$$

Since CS will still be unreadable and M readable, the two parts can easily be separated.

B can obtain S for verification by computing:

$$S \leftarrow CS+e_1 \pmod{n_1}$$

That is, B obtains the message-dependent signature by using A's public key. It is easy to check whether S is a function of M; if so, then only A could have formulated CS since only he has the correct secret key.

Chapter Two

An Improved Secure Communications Network

2.1 Introduction

In this chapter we design a secure public-key communications and authentication network. Our design has three advantages over existing public-key designs by Needham and Schroeder[39], Denning[9], and Michelman[36]:

(1) The entire network, including the central mechanism or directory, can be implemented on small computers, making it inexpensive and portable.

(2) The network has built-in safeguards to provide a measure of protection against cryptanalysis or theft of a user's secret key. Even with a stolen secret key it is difficult for an intruder to sign messages, or otherwise actively use the key, without detection.

(3) Network security breaches are detected more rapidly than in previous designs.

The advantages claimed are realized by distributing the key generation function to the network nodes (thereby decreasing the workload of the central mechanism) and by including a *history buffer* at the central mechanism.

One of the defects of the proposed network is slow operation. The time to initiate logical channels will be relatively large because double encryption of some messages is required and channel capacities will be small because a public-key system must be used. It is shown in Chapter Three that operation will be fast enough for some applications if the RSA cryptosystem is implemented.

Before developing the proposed network in Section 2.4 we first review some secure network concepts in Section 2.2 that will aid in the discussion to follow. Section 2.3 is a summary of the state of the art, as we know it, in secure communications network design, where we define some of the problems inherent to existing networks. Section 2.5 outlines some applications for the proposed secure communications network. Since the network can be implemented at low cost using only software and existing (inexpensive) hardware, numerous applications are evident.

This chapter is partly intended to help dissipate the misconception that:

> "In a public-key cryptosystem, it is necessary only for a central controller to distribute a private key to each user of the system."[51]

We shall show that it is reasonable that the controller never distribute private keys at all.

2.2 Some Secure Communications Network Concepts

A secure communications device (SCD) is an apparatus that allows individuals to communicate more securely than by common carrier alone. SCD's form the nodes in any secure communications network (SCN), which must have a central trusted mechanism which is either a *central facility* (CF)[9] for generation and distribution of user keys, or a *key distribution center* (KDC)[40] for key distribution alone. Since it is generally undesirable that user keys (even public keys) be transmitted openly, any CF or KDC must periodically generate its own key or keypair to encipher user keys before transmission. SCN's use cryptography to create secure logical channels superimposed on physical channels (the common carrier); information can travel on a logical channel at a rate not exceeding the channel capacity.

The requirement for a central trusted mechanism arises from the need for *authentication* or the unambiguous identification of network users. There are two types of authentication[12]:

(1) In *user* authentication, network users identify themselves to each other to establish communications. User authentication can be either *indirect* (in which the fact that a user possesses a key or keypair is used as evidence of his legitimacy) or *direct* (in which

users are identified by personal characteristics such as their voices on telephone lines, their fingerprints, or their handwritten signatures). [1]

(2) In *message* authentication, network users exchange electronic signatures to provide future proof, to an authority, of their acceptance of terms stated in a document.

Since there are no convenient and foolproof methods of providing direct user authentication electronically, direct authentication is used only when a user joins an SCN; at that time an individual furnishes *side information* that proves his identity, such as a driver's license.[2]

If the authentication mechanism of a network fails for some reason, a network user may *repudiate* his signature on a document or his presence on the network at a particular time. Furthermore, if a network user can show that an *intruder* (or unauthorized party) can penetrate the network

---

[1] The necessity for user authentication has been noted by Simmons[46] who relates a protocol devised by Rivest, Shamir, and Adleman for playing 'mental poker'. The implication of the protocol is that persons having cryptographic keys can communicate securely without knowledge of each other's identities. Secure communications without user authentication is unacceptable for most applications.

[2] Side information is also required to join other networks such as credit networks, with means of future indirect authentication (a credit card) being given to the person joining the network.

under some combination of circumstances, then repudiation becomes possible. The best safeguard against repudiation is complete network security.

An intruder may try to gain access to an SCN to obtain information or to alter signed documents and may be *active* or *passive*[13]. A passive intruder eavesdrops on conversations and obtains information through cryptanalysis, whereas an active intruder imitates authorized users or undetectably alters cryptograms. The active intruder therefore requires a legitimate user's key which may be obtained through cryptanalysis or theft. Key theft can occur at the central mechanism or at a physically insecure SCD.

The necessity for a central mechanism leads to a *key distribution problem*: keys must be recorded, transmitted upon request to users wishing to communicate, and replaced in records when they become obsolete. Public-key cryptosystems were designed to alleviate the key distribution problem and simplify the establishment of a secure network.

The central mechanism in a public-key SCN must be extremely reliable for two reasons. First, the safety of the public key scheme depends particularly on the selection of the correct public key for encryption. Also, the maintenance of the directory of public keys is critical because in any SCN design a user's public key will be changed from time to time and this must be done correctly.

Popek and Kline[40] have summarized much of the theory of SCN design. Some of their conclusions are:

(1) Three basic types of SCN are possible, regardless of whether the network uses a public-key or private-key cryptosystem; these are the *fully distributed*, *hierarchical*, and *centralized* SCN's, diagrammed for the private-key case in Figure 2.1. (Public-key SCN's are not diagrammed because their only difference at this level is that they provide <u>two</u> communications paths between any pair of nodes whereas private-key networks have only one path.) Popek and Kline observe that the fully distributed SCN is a variant of the hierarchical SCN and that the centralized SCN is a degenerate version of the hierarchical SCN. When designing an SCN it is therefore necessary only to design a centralized SCN.

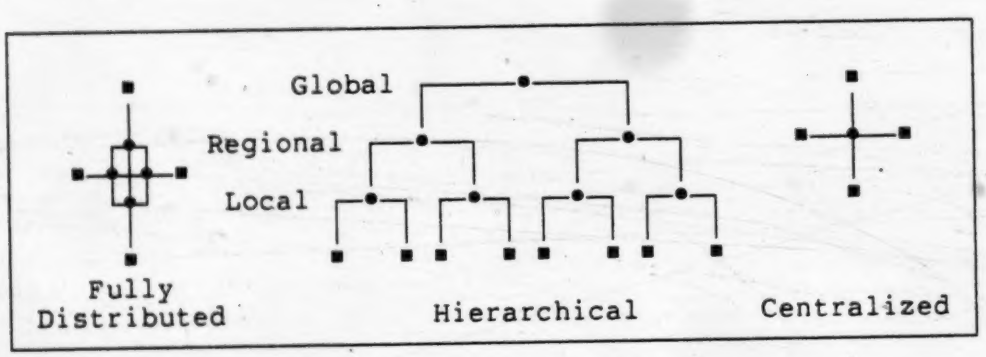

Figure 2.1. Private-Key SCN's (•=CF, ■=SCD)

(2) Key distribution can be either simple or complex depending on the type of SCN. The hierarchical SCN is

useful for very large networks since each CF need only retain a few keys. On the other hand, in the fully-distributed SCN, $n \times (n-1)/2$ matching keys must be arranged among n CF's, with each CF retaining n-1 keys. The CF in a centralized SCN must retain n keys.

(3) Message authentication is possible in any SCN with an appropriately designed central mechanism. *Protocols* for establishing secure channels are outlined by Popek and Kline that allow signatures in both public- and private-key SCN's.

(4) The central mechanism should be minimized to make it reliable and trustworthy. That is, the fewest possible number of persons should have access to the central mechanism, its software should be very reliable, and it should be securely located, perhaps at a computer center supporting a secure operating system.

Popek and Kline place little emphasis on public-key SCN designs in which key generation is done at the SCN nodes. We will show that distributed key generation allows minimization of the trusted mechanism, simplification of key distribution procedures, and increased SCN security.

2.3 Historical Perspective

SCN's have been constructed or proposed but no design to date has been perfect. Existing designs are flawed by such things as their low channel capacities, their expense, their requirement for a very special location for the central mechanism, and their requirement of special unavailable hardware. This section outlines some of the features and drawbacks of existing SCN designs.

Kahn[26] has fully covered the subject of private-key SCN's using mechanical SCD's. The channel capacities of SCN's described by Kahn were very small, but the advent of solid-state electronics and modern telephone lines has permitted the construction of private-key SCN's with large channel capacities, using cryptosystems based on the same principles as those described by Kahn. For example, chip implementations of DES permit inexpensive private-key SCD's to be built, leading to the very high-speed electronic private-key SCD's and SCN's that are occasionally described in Cryptologia. Electronic Funds Transfer Systems (EFTS)[2,20] are a type of SCN; to date EFTS networks use either private-key cryptosystems or no cryptosystem at all.

Despite their potential for large channel capacities, private-key SCN's are flawed in two ways:

(1) Increasing computer power and improved cryptanalytical techniques may make private-key cryptosystems insecure

in the near future.

(2) The central mechanism must generate keys for use by communicating parties. Keys are generated with a *key stream generator*, which is a complex program that must run continually on a large computer to generate the many keys needed by users of the SCN.' The requirement for a large computer and the attendant requirement for a large staff makes private-key SCN's expensive and places constraints on their locatability, since locations with a completely trustworthy staff are uncommon.

There are three proposals that we know of for implementing public-key SCN's:

(1) Needham and Schroeder[39] design a public-key SCN in which all user keys are generated by a CF. A directory of public keys is maintained by the CF.

(2) Denning[9] proposes the conversion of personal computers into SCD's by attaching hardware encryption/decryption units. The Denning SCN requires a CF capable of generating hardware keys to be used in

---

' Key stream generation involves the generation of an ideally non-repeating quasi-random sequence of bits, portions of which may be extracted and used as keys for private-key cryptosystems or as seeds for generation of keypairs in public-key systems.

conjunction with the encryption/decryption units. Key distribution is done by a KDC separate from the CF.

(3) Michelman[36] defines a network requiring the central mechanism to generate its own keys but not necessarily keys for SCD's. The central mechanism communicates with SCN users using its secret key to encipher messages. SCD keys, and those of the central mechanism, are changed only at very specific times.

There are flaws in each of the public-key SCN's enumerated above. One common flaw, of course, is that all three SCN's (including Denning's) have small channel capacities compared to private-key SCN's.

Each system has further flaws. The Needham and Schroeder proposal has a bottleneck created by the requirement that the CF generate all network keys. Centralized key generation for all nodes in a public-key SCN requires a relatively large computer to act as CF (with all the attendant cost, location, and personnel considerations) for even a small network. Furthermore, the key generation program must be a *memoryless subsystem*[16] that does not retain user secret keys after they have been generated and distributed, adding to the difficulty of CF certification.

Denning's proposal is flawed by a bottleneck as in Needham and Schroeder's proposal, as well as two other problems:

(1) Frequent key change is impossible. Because keys are in hardware, an SCN user must physically obtain new keys from the CF. It may be necessary, however, to change keys often: analysis and duplication of microcircuitry is technologically feasible, so a user's key could be stolen, duplicated, and returned surreptitiously.

(2) Hardware public-key encryption/decryption units have been in development for some time, but are not yet available. Whether such units will ever become cheap enough to be generally available is debatable.

Michelman has observed that a flaw with his design is that it is slow in the detection and correction of security breaches; this flaw is common to the other proposals as well. Slow correction of breaches leads to extended access for an intruder who acquires a user's private key, by theft from an SCD or through cryptanalysis. If the SCD's are physically unsecured then it is impossible to guarantee the validity of signatures.

2.4 A Secure Communications and Authentication Network

In the following discussion we design a simple public-key SCD and an SCN showing resistance to key theft and having rapid detection of security breaches. The discussion rests on two assumptions, both justified in Chapters Three and Four:

(1) The channel capacity of a public-key SCN can be made large enough to be acceptable for some applications.

(2) It is possible to distribute the function of keypair generation to the nodes of a public-key SCN, with keypair generation fast enough that users can generate one keypair for each conversation.

2.4.1 A Trivial Public-Key SCD

Under these assumptions the design of a public-key SCD becomes trivial. The only requirements, besides key generation software, are for transmission error recovery software, a modem, and a printer to keep permanent records of signed documents. Such an SCD is completely useless unless it is part of an SCN with a central mechanism, since it is impossible for users to authenticate each other.

It is possible to give some usefulness to SCD's unconnected to a central mechanism by incorporating an ordinary telephone into each SCD to provide some measure of

direct user authentication. Although it is impossible to guarantee the security of communications using SCD's connected only by a telephone line and no central mechanism, it should be remembered that speech synthesis and recognition by computer are difficult, unsolved, problems[33] and that mimickry of human voices by humans is difficult.

The following protocol illustrates how two SCD users, A and B, can initiate a (somewhat) secure logical channel, each knowing only the other's voice. The security of their communication is entirely dependent on the reliability of voice recognition over a telephone line.

(1) A and B authenticate each other by voice, using the telephone.

(2) A sends his public key to B. B sends his public key to A.

(3) A generates a random number and concatenates this number with a short message. The resultant text is then encrypted using B's public key and the resulting cryptogram is transmitted to B. User B carries out the same sequence of actions as A. That is, B generates a random number, concatenates it with a short message, encrypts the result with A's public key, and sends the resulting cryptogram to A.

(4) After A and B have deciphered the cryptograms, the resulting plaintext messages are relayed back to the original senders via the telephone voice links.

(5) If both A and B feel confident that they have received each other's keys, then they subsequently use these keys for enciphering messages.

(6) As an additional precaution, all messages sent from A to B are tagged with the random number generated by B. Furthermore, all messages sent from B to A are tagged with the random number generated by A.

2.4.2 An Improved KDC and Protocol

a. The KDC

The proposed KDC is diagrammed in Figure 2.2. Some memory is required for the *history buffer* which is divided into columns, one column for each network user. Each column can record n user keys, where n is the maximum number of keys allowed any user in a particular period of time; it might be decided, for example, that users are permitted 20 conversations (and therefore keys) in one day. The limitation on the history buffer size depends on available memory, number of network users, and frequency of network use desired by the users.

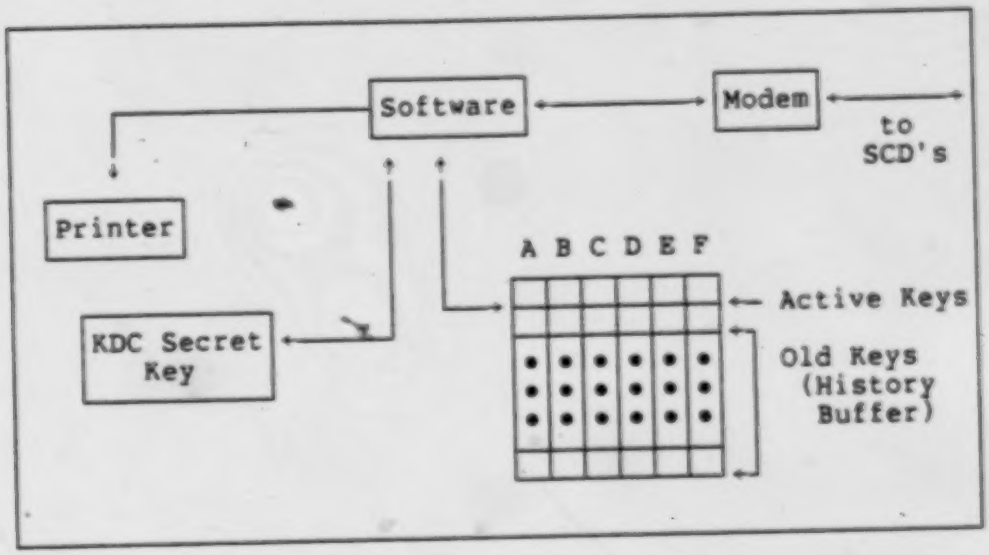

Figure 2.2. The Proposed KDC (6 Users)

The KDC software includes software for key generation, encryption/decryption, command interpretation and key distribution, and transmission error recovery. The software is certified[10].

In any SCN design the central mechanism must be able to unambiguously identify itself to SCD's. Our KDC identifies itself in the same way as in previous SCN designs: it generates a keypair and encrypts all outgoing messages with its secret key. Its public key is available to everyone. Let (KDC,P) be the KDC's public key, which may be openly published, and let (KDC,S) be the KDC's secret key, which is known only to the KDC. We set no specific time constraint on KDC key changes; it may change keypairs as often as convenient.

b. User Initiation

To join our SCN, an individual first identifies himself to a trusted KDC operator (direct user authentication) and gives the operator a public key generated at the individual's SCD, perhaps written on a piece of paper. The operator enters the key at the KDC console and gives the new user the KDC public key (or tells him where the KDC key is published). The KDC may have simple software to test that the key submitted does not match any other user's key in memory; the possibility of a match is very remote. The user's key is placed on top of his history buffer column.

c. The Commands

Each user has just two commands that he can send to the KDC, "REQUEST" and "RELEASE":'

(1) *REQUEST*. For Y to obtain X's public key, Y must REQUEST it by sending

Y←F{'REQUEST X',(Y,S)}

to the KDC. 'Y' is a plaintext identifier indicating to the KDC the public key that must be looked up to decipher the command; that is, (Y,P). Although anyone

---

' The notation 'F{M,(Y,S)}' in what follows means: "Using the public key cryptosystem implemented, use encryption function F to encipher message M using Y's Secret key." '←' in what follows means string concatenation.

having Y's public key can decipher the command, no security breach occurs if this happens; what is important is that the KDC knows that only Y could have sent the message.

(2) *RELEASE*. Before the KDC can transmit X's public key to Y, X must RELEASE the key by transmitting

X+F{F{'RELEASE X'+(X,P[new]),(X,S[present])},(KDC,P)}

to the KDC. The RELEASE command requires that X generate a new keypair to be used for his next conversation; the present key is used for the present conversation only. The command requires double encryption because only the KDC is to know X's new key (hence the use of (KDC,P)) and the KDC must be sure that X is the transmitter (hence the use of (X,S[present])).

d. The Protocol

Two SCD users, A and B, establish a secure logical channel using the following protocol:

(1) A REQUEST's B's public key from the KDC. The KDC prints the encrypted and decrypted versions of the REQUEST for future proof that the request was made.

(2) The KDC transmits

KDC+F{'RELEASE TO A?',(KDC,S)}

to B. Although anyone can decipher the KDC's transmission, no security breach is involved. B can be certain the transmission originated with the KDC.

(3) If B decides that communication with A is desirable, he generates a new keypair and transmits a RELEASE command to the KDC. The KDC decrypts with (KDC,S), prints the result, decrypts again with (B,P[present]), and prints the plaintext result. If (B,P[new]) matches any other key in B's column of the history buffer, the KDC signals 'SECURITY THREAT' and terminates operation.

(4) The KDC obtains (B,P[present]) from the top of B's column of the history buffer and transmits

KDC+F{F{'B's KEY:'+(B,P[present]),(A,P)},(KDC,S)}

to A. Double encryption is needed to assure A that the KDC transmitted the message and to ensure that only A can decipher the message. The KDC places (B,P[new]) on top of B's column of the history buffer, pushing (B,P[present]) to the second position, and all other old keys down one position.

(5) Steps (1) to (4) above are repeated with 'A' and 'B' interchanged for B to receive A's public key. A and B now have each other's keys and can communicate securely.

e. Discussion

The network's security depends on the trust that can be placed in the KDC's keypair, which is used to encipher numerous small messages; the keypair should therefore be of high quality. If desired, time and date stamps can be added to messages sent by the KDC for additional security.

Although the keys in KDC memory are called 'public' keys they are not handed out freely to anyone who wants them, in accord with Michelman's dictum that there is no security in a network in which keys are given out without restraints.

Since user keys are changed frequently they may be made relatively short and still provide excellent security. The printed record of all key changes provides a log that pinpoints the time of each conversation or signature and the public keys used, thus localizing security breaches. The use of ever-changing keys for signatures is a discrete analogue to the continuous, very slow, changes in ordinary handwritten signatures.

The history buffer is used to protect network users against key loss or theft in the following way. Assume that a user leaves his SCD signed on and unguarded for a short time and an intruder copies the user's secret key from SCD memory. If the intruder attempts to use the key at his own SCD he must replace it on top of the history buffer when RELEASE'ing it. When the legitimate user attempts to use his key, he will find that it is no longer valid because it

matches an old (i.e., pushed-down) key in the buffer; the KDC will signal 'SECURITY THREAT' and not allow the legitimate user to communicate. The intruder may attempt to cycle the history buffer to return the key on top to its original value by having *n* conversations with network users; however, if *n* is made large and the time constraint on the buffer is long, it is unlikely that the intruder can cycle the buffer before the legitimate user attempts to use his key.

Of course, an intruder can *eavesdrop* on one end of a conversation with a stolen key; the history buffer only prevents active use of a stolen key. Note, however, that eavesdropping can only be done for <u>one</u> conversation and that a stolen key is of no value in determining the user's next secret key.

The proposed SCN is vulnerable to the threat of *theft and replacement*; that is, it is possible for an intruder to copy a user's key, replace it with another in the user's SCD's memory, use the stolen key at his own SCD, and replace it on top of the history buffer with a key matching the one placed in the legitimate user's SCD. The only defence against this threat is that the user either memorize his secret key or write it down. Even memorization of part of the key should suffice to keep it from being replaced undetectably, but the responsibility is the user's.

2.4.3 Further Considerations

There are two broad areas in which the proposed SCN could be improved: (1) Additional Security, and (2) Versatility and Dependability.

a. Additional Security Measures

(1) We have shown that it is possible to automatically guard against short-term loss of control of the SCD. It is the user's responsibility, however, to prevent an intruder from gaining access to the SCD or its software for an extended period. An extended period of intruder access may allow the introduction of a *Trojan Horse*[30]; that is, modification of the SCD's software and possibly hardware so that information is transmitted in the clear or, perhaps, so that weak keypairs are generated. There are a number of measures that can be taken to guard against the introduction of a Trojan Horse, including secure storage of the disk and SCD after use, occasional visual verification of the source code and recompilation, and perhaps construction of software to provide a checksum of both the old and new object decks to force an intruder to be extremely subtle in introducing modifications.

(2) There are measures in the broad area of *data security*[10,11,19,44] to aid in safeguarding keys and

software. Some of these are implementable on a small computer; for instance, a strong password scheme could be implemented involving the use of a rapidly changing value such as the time of day.

(3) A precautionary measure entailing some expense involves the use of a 'water marked' magnetic card and a magnetic card reader/printer. Such a device and card would be used to split a secret key into two parts at signoff, with one part left in secondary storage and the other on the card. With private-key cryptosystems this splitting of a key requires the generation of a quasi-random number that is stored on the card and subtracted from the key in memory[7]. Public-key schemes allow some simplification of this procedure; for instance, if the RSA cryptosystem were implemented the secret key d could be stored on the card with n left in memory.

(4) Even if the key at each SCD cannot be made perfectly secure, an additional mechanism, called a *(k,n)-threshold scheme*, can be placed on the KDC to guard against forgeries. In such a scheme k users must cooperate to sign a document. Shamir[43] describes a (k,n) scheme requiring the use of passwords. A (k,n) scheme could be easily implemented on the proposed SCN by simply requiring k REQUEST's and k RELEASE's before authentication could proceed. Individual users might

be given more signature authority than others by storing a weight factor along with their keys at the KDC.

b. Versatility and Dependability

A number of improvements distinct from increased security can be built into the SCN to make it more convenient, more applicable, or more reliable. Some possible improvements are:

(1) Throughout this chapter half duplex operation has been assumed for simplicity, in that only one end of a logical channel was expected to be transmitting at any time. Additional mechanism in the SCD's and KDC would be needed to provide full duplex operation, but implementation should be straightforward.

(2) It has been assumed that *packet switching*[40] is not required; that is, a direct physical channel is assumed to exist between communicating users and messages do not have to be switched from one KDC to another. A packet switching mechanism would be needed to allow SCN's to communicate, necessitating an extra layer of mechanism in the KDC's. Messages would need identifying labels and special formats[7].

(3) A mail system could be implemented with additional mechanism in the KDC and SCD's. Some complications are

apparent because of the requirement for active RELEASE of public keys by their owners. A third command, 'MAIL', might be added that would not require active key release; alternatively, users might generate special mail keys which would not be protected by the history buffer.

(4) For additional channel capacity, a *hybrid system*[12] could be implemented; that is, one in which users can encrypt material using a fast private-key cryptosystem such as DES, with the public-key algorithm used to pass DES keys. Generation and storage of DES keys would have to be entirely SCD-bound to avoid the necessity of a large central computer, so although some modifications to the SCD's would be required the KDC would need none.

(5) *Redundancy* of the KDC and printer is required to prevent partial or complete communications failure in case of a breakdown[40]; additional software would be needed in the multiple KDC's to simultaneously update multiple history buffers and handle multiple printers. Redundant KDC's would allow faster network operation, of course, while they were operational.

(6) If the RSA cryptosystem is implemented faulty keypairs will occasionally be generated which may not be easily detected by user testing. It is easy to design a

protocol for replacement of faulty key if each SCD always retains the two secret keys most recently generated.

2.5 Applications

Since the proposed SCN can be implemented entirely on small computers, it will have more applications than previous more expensive designs. Some possible applications include internal business communications, transmission of prescriptions from doctors to pharmacists, electronic voting and census (decryption keys would be obtained by enumerators), electronic notarization (including recording of patents and copyrights), invoicing (i.e., business to business or KDC/KDC communications), electronic funds transfer, transfer of securities by brokerage houses, and ordinary interpersonal use, eventually possibly the most important application of all.

Since the keypairs act as *capabilities*[14] or tickets conferring privileges on the key holders, the system might be used for interprocess communication and synchronization if implemented on large computers. If a logical channel is thought of as a resource that is shared by processes then it can be seen that process deadlock on a resource is impossible using our protocol and security in a distributed system is guaranteed. Damage to a resource can be traced

back to the process that did the damage.

An example by Whiteside[49] emphasizes the necessity for an SCN for internal business communications in at least some industries. Whiteside observes that a large oil company lost many millions of dollars by being underbid by a competitor for tracts in Alaska. Information belonging to the oil company was apparently intercepted by the competitor while enroute between Alaska and New York. The proposed SCN would have minimized the possibility of such interception at a cost of only a few thousand dollars for software, small computers, and modems.

2.6 Summary and Conclusions

It has been shown that a secure communications network can be constructed at low cost, if it may be assumed that a public-key cryptosystem can be implemented on small computers. The network is highly resistant to penetration by outsiders and repudiation by legitimate users. The network departs from previous proposals in that only small computers are needed, users generate their own keys for each conversation, and a history buffer is included in the central mechanism to make it difficult for an intruder to actively use a stolen secret key.

In part, this chapter has been meant to aid in discussion of the interesting question, "How secure and impenetrable can we make a communications and authentication network, given the probably secure public-key cryptosystems now available?" After all, it seems futile to design a cryptosystem requiring an intruder to do thousands of years of cryptanalysis to obtain a key, if he can simply steal the key from an unsecured communications device. The proposed solutions are a first step in answering the question.

We do not claim that the proposed network is impervious to all threats by a resourceful and knowledgeable intruder, nor that repudiation is completely impossible. Some forgeries may still occur. It should be kept in mind, however, that even handwritten signatures can be, and occasionally are, forged.

Chapter Three

RSACRYPT: An Implementation

3.1 Introduction

RSACRYPT implements the RSA cryptosystem on an AMDAHL 470V/8 computer at the University of Alberta. The program was constructed for two reasons:

(1) To determine the difficulties, if any, that must be overcome before the RSA cryptosystem can be implemented on microcomputers, and

(2) To predict the speed of RSA keypair generation attainable on a microcomputer.

In this chapter we first briefly describe the program in Section 3.2. The difficulties faced and overcome in implementation are then discussed in Section 3.3. Finally, in Section 3.4 we relate the results obtained to those of Michelman[36] to predict the performance of the RSA cryptosystem on the MC68000 microcomputer and to reach some conclusions about the practicality of the network proposed in Chapter Two.

3.2 The Program

It seems that modularity has been discussed only in the context of private-key cryptography[4]. However, an implementation of the RSA public-key cryptosystem also requires that very distinct modules be implemented separately.

RSACRYPT consists of seven major modules and three modules peripheral to the RSA cryptosystem. Coded almost entirely in ALGOL68, RSACRYPT is designed to be easy to understand and use and to be user modifiable for additional cryptosecurity; its operation and modular nature are depicted in Figure 3.1 (called modules are bracketed). The seven major modules are organized into three groups as follows.

Group 1: Global Modules

(1) SERVICEPAK is a utility package that carries out functions such as the opening and closing of files for interaction with the user, for key storage, and for encryption and decryption of messages.

(2) MATHPAK is a multi-precision arithmetic package.

(3) DESPAK contains all the procedures and tables required for a software implementation of DES. It is called with a string, the first 8 characters of which are

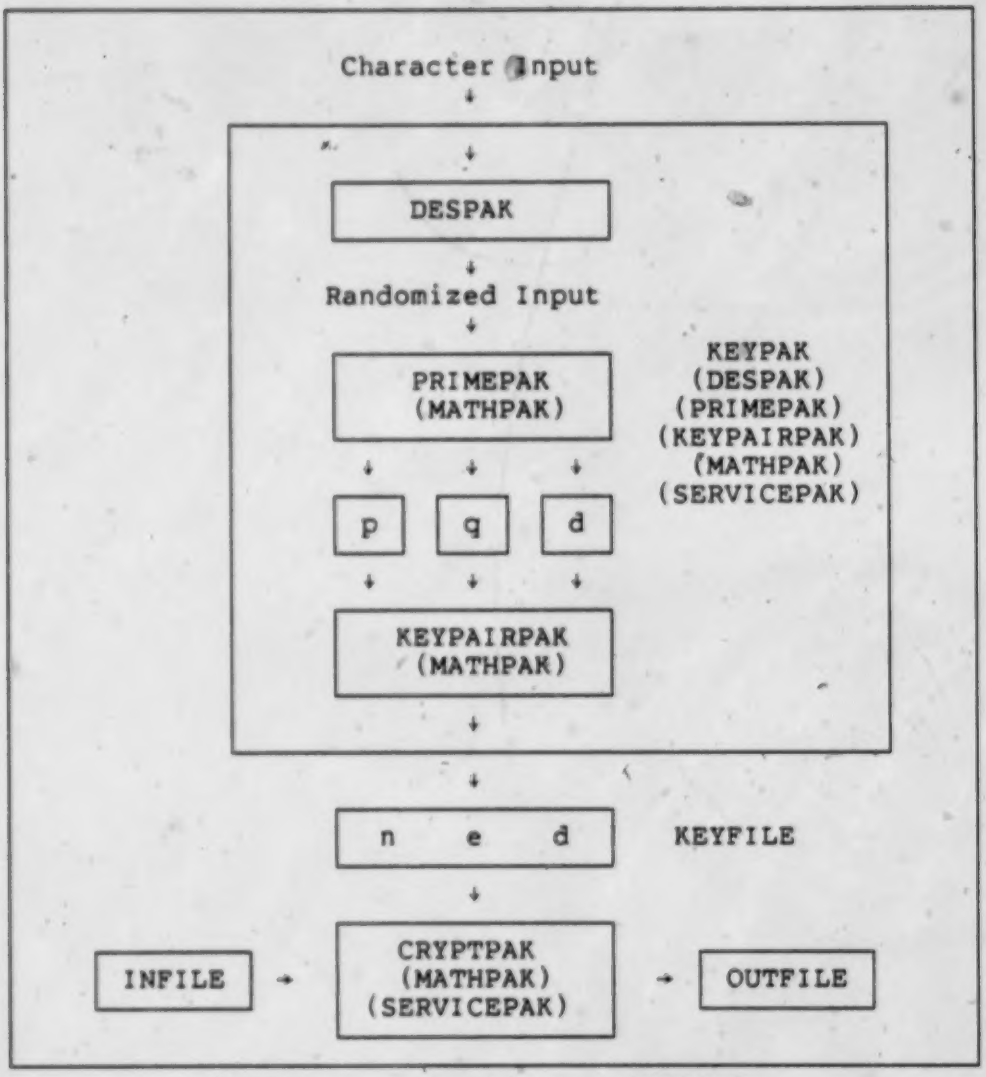

Figure 3.1. Operation of RSACRYPT

treated as a 64-bit DES key, and returns a string that can be used as a quasi-random number. It uses block chaining for increased resistance to cryptanalysis.

Group 2: Keypair Generation Modules

(4) PRIMEPAK returns, for any given integer, the next prime greater than or equal to the integer. The number returned is *probabilistically prime*; the probability can be made arbitrarily close to unity through user input of the amount of primality testing desired.

(5) KEYPAIRPAK is passed 3 prime numbers and uses MATHPAK to obtain an (e,d,n) triple.

(6) KEYPAK interacts with the user and determines the length (in bits) of the keypair to be generated, invokes DESPAK to scramble an input from the user, and transforms the quasi-random number thus obtained into a keypair, with calls to PRIMEPAK and KEYPAIRPAK. The keypair is written to a user-specified file.

Group 3: Encryption/Decryption Module

(7) CRYPTPAK encrypts or decrypts a message in a user-specified file. It interacts with the user to determine the key to be used.

The three peripheral packages mentioned above include one to allow the user to choose between a key generation run and an encryption/decryption run, and two assembly language routines to allow visual editing of files and reassignment of logical I/O units without unloading the program.

Because of its modular nature RSACRYPT should readily lend itself to future improvements. For instance, the program owner could easily detach DESPAK as used for quasi-random number generation and attach a quasi-random number generator of his own design, or if a faster prime number generation scheme is devised PRIMEPAK could be replaced. As well, RSACRYPT should serve as an excellent basis for implementing other public-key schemes: any public-key cryptosystem may be expected to require a mathematical package, a random number generator, a keypair generator, an encryption/decryption package, and perhaps a prime number generator.

Since the program is large and complex some measures were taken to permit verification by the user and possibly certification. The data structures used were kept simple: a few vectors, representing the integers p, q, n, e, d, and the Euler totient, are manipulated and transformed as required. Procedures were written with emphasis on clarity and documentation; most procedures have test drivers attached.

A knowledgeable user could be expected to gain some understanding of the program in a matter of a few hours. Even a rudimentary understanding would allow him to at least verify that the program does not write out a plaintext version of his message other than to the file specified. At a deeper level, he could verify that the program carries out the RSA and DES specifications precisely, with no

deviations, or satisfy himself that the program is a memoryless subsystem that does not retain the prime factors p and q after a key generation run, for example.

3.3 Difficulties

3.3.1 Quasi-Random Number Generation

The generation of good quasi-random numbers is of critical importance to an implementation of the RSA cryptosystem since if the seeds for generation of p and q are insufficiently random a cryptanalyst might find a way to factor even a very long n. There are many methods for generating quasi-random numbers, ranging from trivial to highly complex; Knuth[29] discusses much of the theory. In this application a method is needed which is compact enough to allow the code to be easily implemented on a small computer, which still provides sufficient resistance to cryptanalysis.

Two extreme examples of generators considered and rejected for this application are the *mid-squares* method used by Von Neumann and mentioned by Knuth[29], and the highly sophisticated TLP generator designed by Bright and Enison[3]. The mid-squares method involves the repeated squaring of a seed and extraction of the result's middle digits; the method is certainly compact but its simplicity leads to doubts about its ability to resist concerted

cryptanalysis. On the other hand, the Bright and Enison method is anything but compact: it generates large tables and is suited to keystream generation on a large computer but not to quasi-random number generation on a small computer.

It was decided to use a software implementation of DES to scramble a seed input by the user; this decision was made because it has been observed that strong private-key cryptosystems are by definition excellent quasi-random number generators[3,46]. The seed used is of the same length as the sum of the lengths of the three quasi-random numbers required (plus 64 bits for the DES key) and is entered as a string that, for practical keypairs, is at least 100 characters long. A length of 100 characters or more is probably enough to prevent the user from introducing an unconscious bias into the seed that would permit cryptanalysis. As an additional precaution, block chaining is used to scramble the seed even more than is possible by simple use of DES; the number of rounds of chaining is user-specified. The quasi-random number generated is split into three parts that are used as seeds to generate p, q, and d for the RSA keypair.

We believe that DESPAK overcomes the shortcomings of DES because this is a software implementation so the program owner has access to the DES tables. Since changing even one bit in any of the tables will radically alter the ciphertext obtained from a given message/key combination, the owner can

ensure that the hypothesized DES trapdoor cannot exist.

Additionally, in this application the quantity of DES-processed text is small (a large quantity is necessary for cryptanalysis) and unavailable to a cryptanalyst since it is further disguised by transformation into an (e,d,n) triple.

One shortcoming of DES that DESPAK does not resolve at present is the short key, but the key length could easily be increased if deemed necessary. Even more security would be provided by increasing the amount of input by the user and discarding some of the scrambled output before use for keypair generation. We believe that a 64-bit key is probably ample, however, for the short 'messages' encrypted.

3.3.2 Prime Number Generation

The method used for prime generation is the probabilistic method outlined by Rivest, Shamir and Adleman[42]. This method involves the use of Fermat's Theorem: for a prime number p and any integer a < p it is always true that

$$a^{(p-1)} \equiv 1 \text{ (modulo p)}.$$

A number p is tested for primality by applying Fermat's Theorem repeatedly with a number of different a's. If k tests are passed then p is prime with probability

$$1 - 1/(2^k).$$

If p fails a test then it is incremented by 2 and testing begins anew. Rivest, et al. believe that on average

approximately 150 candidates will be tested to obtain primes of the recommended sizes.

Obviously, the generation of primes using Fermat's Theorem can be done with the same routine for modular exponentiation as used for encryption, with the only difference being that in encryption the modulus is constant for a number of message blocks whereas in prime generation p changes frequently. Therefore, a hardware encryption/decryption unit, when one is built, can be used to rapidly generate keypairs as well as encipher messages.

Two questions occur when programming this algorithm:

(1) How should the values of *a* be chosen?

(2) Modular exponentiation is a time-consuming process. Can its use be avoided to some extent when generating primes?

In answer to the second question, the use of modular exponentiation has been reduced by approximately a factor of 5 in the following fashion.

Given any odd positive integer, p, one easy test that rejects 33% of all non-primes is to divide p by 3. This idea can be extended to division by 5, 7, 11, and other small primes. Division by the first 8 small primes rejects approximately 80% of all odd numbers as possible primes. (Extending the list beyond 8 primes will only increase performance slightly.)

To avoid dividing every candidate for primality by all 8 prime divisors, a count is associated with each divisor that is initialized to zero when the divisor is found to evenly divide any candidate. A count that has been initialized is simply incremented modulo its associated divisor for all succeeding candidates. After only a few candidates have been tested in this fashion all counts are initialized and division by prime divisors is entirely eliminated. Henceforth, all counts are incremented for succeeding candidates and whenever all the counts are non-zero modular exponentiation is used for further testing.

Returning to the question of how the values of *a* should be chosen, Rivest, et al. suggest that random values be used. This suggestion need not be taken literally since all that is required is an unbiased set of *a*'s that gives each candidate as fair a test as possible; furthermore, in practice, the generation of quasi-random numbers is far too time-consuming to generate the many required values of *a*.

RSACRYPT generates *a*'s rapidly and with very little memory requirement in the following way. A short list of digits is entered as randomly as possible by the programmer before compilation. During a run, *a* is always initially set to 3 as recommended by Knuth[29]. As successive values of *a* are needed, digits from the short list are prepended to the current value of *a*. If prepending alone were used, however, *a* could grow larger than p, which is unacceptable because it is unnecessary and slows down the generation of the primes;

therefore, when many tests of primality are to be done an increment is computed and *a* is sometimes incremented by the computed amount without prepending. In this way, the many values of *a* are folded into a small amount of memory. Since the user specifies the amount of testing to be done, which determines the increment, the programmer has little control over the *a*'s that are generated; in a sense, they are quasi-random.

3.4 Analysis of Results

Recall that modular exponentiation is used for both encryption/decryption and key generation in the RSA cryptosystem. Modular exponentiation requires the repeated use of multiplication and division (see Chapter Four) and is $O(n^3)$ if the standard, $O(n^2)$, algorithms for multiplication and division are used.

The results obtained from RSACRYPT, combined with Michelman's results[36], permit the prediction of the performance of the RSA cryptosystem on microcomputers. For brevity we derive timing estimates for only the MC68000.

3.4.1 RSACRYPT Results

RSACRYPT encrypts, decrypts, and generates keypairs very slowly because of the use of ALGOL68 and a radix of 256 to represent integers, as well as the use of the standard

algorithms for multiplication and division.[1] Table 3.1 illustrates the encryption speed of RSACRYPT for various sizes of the modulus n, with the key (e or d) the same length as n.

n Size (decimal digits)	Bits Encrypted per second
20	40.0
37	11.8
51	7.6
73	3.5
155 (512 bits)	0.32 (projected)
200 (660 bits)	0.13 (projected)

Table 3.1. Encryption Speed of RSACRYPT

The first 4 values in Table 3.1 were obtained by actually running the program, permitting the solution of 4 equations in 4 unknowns to obtain a timing formula, which was used to obtain the last two (projected) table entries. The timing formula for encryption/decryption is (in seconds/character):

$$T = -0.805 + 0.173n - (1.927/255)n^2 + (0.034/194)n^3$$

where n is the number of radix 256 digits in the modulus. (RSACRYPT uses a radix of 256; Table 3.1 is in decimal digits for convenience. The conversion between the two

---

[1] If M(m,n) is the time required to multiply an m-digit with an n-digit number (radix 256) and if D(m,n) is the time required to divide an m-digit by an n-digit number, we have found that the timing formulas for multiplication and division in RSACRYPT are:

$$M(n,n) = 27 + 36n + 62n^2 \quad \text{(microseconds)}$$
$$D(2n,n) = 1230 + 532n + 56n^2 \quad \text{(microseconds)}$$

radices is done by assuming that 3.3 bits is needed to represent a radix 10 digit and 8 bits to represent a radix 256 digit.)

Since prime generation makes use of modular exponentation it too is $O(n^3)$, permitting the same technique to be used to obtain a timing formula. Table 3.2 shows the times for key generation for various key sizes.

n Size (radix 256 digits)	Time to Generate Keypair (seconds)
37	60
44	100
51	130
73	510
155 (512 bits)	6689 (projected)

Table 3.2. Key Generation Time of RSACRYPT
(5 Tests of Primality; Ordinary Primes)

The first 4 entries allow the derivation of a timing formula which is (in seconds):

$$T = 53.1 - 4.05n - 0.256n^2 + (592/19342)n^3$$

where n is the number of radix 256 digits in the modulus. The last entry in Table 3.2 has been obtained using the above formula.

Two things must be noted about table 3.2:

(1) The primes generated to form keypairs each had only 5 tests of primality (that is, 5 a's). Rivest, Shamir, and Adleman recommend 100 tests of primality.

(2) The primes generated do not conform to the Rivest, et al. recommendation that primes, p, be generated such that p-1 has a large prime factor, u, and that u-1 also has a large prime factor, v. The generation of such primes p can easily be accomplished by first generating a prime v, doubling it and adding 1, finding the next prime u, doubling u and adding 1, and then finding p.

3.4.2 Michelman's Results

Table 3.3 duplicates the relevant results obtained by Michelman.

Machine/n Size	Key Size = n (bits encrypted per second)
16-bit Machine Size(PDP11)	
200-decimal-digit n Size	
without cache(PDP11/45)	13
with cache(PDP11/70)	30
100-decimal-digit n Size	
without cache(PDP11/45)	47
with cache(PDP11/70)	106
32-bit Machine Size(370/168,cache)	
200-decimal-digit n Size	542
100-decimal-digit n Size	1880

Table 3.3. Michelman's Results

Michelman's implementation is 1.5 times faster than an implementation using the standard algorithms coded in assembly language would be, because he uses a variant of the Karatsuba technique for multiplication that allows multiplication to be done twice as fast as with the standard

algorithm and he uses standard division to obtain remainders. Michelman observes that his results are not the best that could be obtained: no special coding tricks were used.

3.4.3 Projected Encryption Speed on the MC68000

Michelman's results for the PDP11/45 can be used to derive Table 3.4, a table of expected encryption speeds on the MC68000 assuming that the same algorithms as Michelman used are used for multiplication and division.

Machine/n Size	Key Size = n (bits encrypted per second)
16-bit Machine Size(MC68000,no cache)	
200-decimal-digit n Size	5.2
100-decimal-digit n Size	18.8

Table 3.4. Projected Encryption Speeds on the MC68000 (Same Algorithms as Michelman)

Table 3.4 was derived using the following assumptions:

(1) The overhead of multiplication and division on the MC68000 (i.e., indexing, etc.) will be the same as on the PDP 11/45, everything else being equal. That is, if the MC68000 could do a 16-bit multiplication at the same speed as the PDP 11/45, modular exponentiation would be done at exactly the same speed as on the PDP 11/45. Given the improved architecture of the MC68000 this assumption is expected to be conservative.

(2) The PDP 11/45 used by Michelman took 3.5 microseconds to do a 16-bit unsigned integer multiplication. There are various possible multiplication speeds on a PDP 11/45 depending on the type of memory used. 3.5 microseconds is an intermediate value (PDP 11/45 Processor Handbook).

(3) The MC68000 with an 8Mhz clock takes 8.75 microseconds to do a 16-bit unsigned integer multiplication (MC68000 User's Manual).

3.4.4 Projected Key Generation Speed on the MC68000

We can now estimate the time needed to generate keypairs on the MC68000 with the same algorithms as used by Michelman. Since a number of assumptions have been made, and since we have not verified Michelman's results, the estimate derived may be somewhat in error.

From Tables 3.1 and 3.2 we see that the time to generate a 512-bit key using RSACRYPT is projected to be 6689 seconds and the encryption rate using a 512-bit key is projected to be 0.32 bits/second. The encryption speed on the MC68000 using a 512-bit key is derived to be 12.8 bps (by using Table 3.1 to compute the ratio of encryption speeds using 200-digit keys and 155-digit keys, and then using this ratio to compute the rate for 155-digit keys on the MC68000 by multiplication with the entry for 200-digit keys in Table 3.4). Therefore, we anticipate that generation of 512-bit

keys on the MC68000, using the same algorithms as used by Michelman, will take

$$T = (0.32 \times 6689) / 12.8 = 167 \text{ seconds} = 2.8 \text{ minutes}.$$

The generation of primes, p, with p-1 having a large prime factor as described previously, will take approximately 3 times as long as the generation of ordinary primes. Only the factors of n (i.e., p and q) need be generated in this fashion; d can be an ordinary prime. Therefore, the time to generate a keypair in the recommended fashion will increase by no more than a factor of 2.33. In fact, the increase will be less than a factor of 2.33 since the d's generated with RSACRYPT were 1.5 times as long as p and q and took a larger proportion of total key generation time than p or q. Therefore, we conservatively estimate that the generation of keypairs in the recommended fashion will take twice as long as derived above, or 5.6 minutes on an MC68000.

If each prime is tested 100 times for primality as recommended by Rivest, et al., it can be expected that generation of a 512-bit keypair will take (100/5)(5.6) = 112 minutes or 1.9 hours most of the time, since only 1 of 32 primes generated using 5 tests will be rejected by further testing; occasionally the time to generate a keypair will be substantially greater than 1.9 hours.

3.4.5 Projected Rates With Improved Algorithms

In Chapter Four we show that for 512-bit keypairs it is possible to carry out modular exponentiation nearly twice as fast as by using the algorithms used by Michelman. Using our improved algorithms on the MC68000, keypairs will be generated in 2.8 minutes using 5 tests of primality, 56 minutes using 100 tests, and messages will be enciphered at approximately 25 bps using 512-bit keys.

A fast typist types at 60 words per minute, which is 40 bps if a word is assumed to be 5 characters long (8 bits/character). Thus, if the improved algorithms described in Chapter Four are implemented it will be possible to encrypt at a rate exceeding the speed of a better than average typist, with 512-bit keys which will provide ample cryptosecurity (see Table 1.1).

A key generation time of 56 minutes is too slow for application in the network proposed in Chapter Two. In practice, however, we feel that it is unnecessary to test each prime 100 times since users must be able to easily change faulty keypairs in any case. We recommend that between 10 and 20 tests of primality be made for each prime. Since 7 primes are generated for each keypair (3 each for p and q), keypairs will have a probability of 1/146 of being bad if 10 tests are used. Generation time will usually be 5.6 minutes using 10 tests, enabling SCN users to have a maximum of 85 conversations every 8 hours, which is ample.

Every two or three 8-hour days users can expect to generate a faulty keypair.

3.4.6 Projected Network Rates

Recall that in the protocol designed in Chapter Two some messages are enciphered once and some are enciphered twice. Assume that keys are 512 bits long and that encryption speed is 25 bps. With these assumptions, single encryption of a 512-bit block takes 20.5 seconds and double encryption of two blocks' takes 82 seconds. For A to receive B's key, A enciphers an outgoing message once and deciphers an incoming message twice, for a total of 102.5 seconds. B also takes 102.5 seconds to transmit his key. The KDC does twice as much work as either A or B; thus the KDC takes 205 seconds. Equivalent amounts of time are needed for B to receive A's key. Further, A and B must each take approximately 5.6 minutes to generate keypairs with 10 tests of primality.

Therefore, A and B each need 9 minutes of computation to exchange keys and the KDC takes 7 minutes to effect the exchange. In other words, network users are limited to a maximum of 60/9 conversations per hour (null conversations), or approximately 6.7, and the KDC can establish a maximum of 60/7 = 8.5 logical channels per hour. In 8 hours two users

---

[1] Two blocks of 512 bits are needed to encipher a 512-bit key (e) and the other information transmitted with the key. The modulus n can be openly transmitted since it is useless without the key.

will actually be able to have only 53 conversations, not the 85 estimated earlier, because of the limitations imposed by the protocol.

3.5 Conclusions

It can be seen from study of Figure 3.1 that only relatively small portions of RSACRYPT need be in a computer's main store at any time. It should therefore be possible to implement the RSA cryptosystem on a computer with limited main memory.

The factor of 5 increase in speed of prime number generation obtained by using a list of prime divisors for preliminary testing is seen to be significant; otherwise, keypair generation would not be practical in the SCN designed in Chapter Two.

Chapter Four

Towards Faster Modular Exponentiation

4.1 Introduction

A practical implementation of the SCN designed in Chapter Two, using the RSA cryptosystem on microcomputers, requires an algorithm for fast modular exponentiation, both to permit a convenient channel capacity and to enable users to generate keypairs frequently.

In this chapter we first show, in Section 4.2, how modular exponentiation is done using only repeated multiplication and division (to obtain remainders), implying that one way of speeding up modular exponentiation is to use fast algorithms for multiplication and division. Since the aim is to develop a practical microprocessor-based SCN, however, algorithms that are asymptotically fast may not be suitable for improving the speed of modular exponentiation in practice because of high timing constants or for other practical reasons. Section 4.3 is a brief discussion of several possible methods for improving the speed of modular exponentiation that we have found infeasible without further research.

In Section 4.4 we develop methods for multiplication and obtaining the remainder of a division that enable modular exponentiation to be done 3 times faster than by using the standard algorithms for multiplication and division. The

method derived for multiplication is basically an extension of previous ideas, involving nonrecursive use of the Karatsuba technique, and is a practical method that will work in the range of numbers used by the RSA cryptosystem. The algorithm designed for finding remainders, on the other hand, is a general algorithm that can be used in conjunction with any fast multiplication algorithm to obtain remainders with the same time complexity as the multiplication algorithm used.

Although it appears that microcomputers exhibiting some degree of parallelism will not be available for some time, a parallel algorithm that is evident from study of the Karatsuba algorithm is also presented in Section 4.4.

4.2 The Modular Exponentiation Problem

Modular exponentiation is the computation of

$$m \uparrow e \text{ (modulo } n\text{)}$$

for m, e, n integers. In the RSA cryptosystem m (the message) and e (the key) are less than n.

The procedure recommended by Rivest, Shamir, and Adleman[42] for carrying out modular exponentiation, called *exponentiation by repeated squaring and multiplication*, is discussed by Knuth[29]. The basic algorithm is (from Michelman[36]):

```
c := 1;
for i to log₂(e)
do
   c := rem((c*c),n);
   if e(i) = 1 then c := rem((c*m),n) else skip fi
od;
```

The rem operation is done to keep numbers in a manageable range in practice. The result is congruent (modulo n) to the result that would be obtained by simple exponentiation. Note that, if m and n are d digits long, then after each multiplication or squaring c will be 2d digits long and will again be d digits long after the rem operation.

Exponentiation by repeated squaring and multiplication is essentially a method of evaluating powers by grouping multiplications so that fewer multiplications are carried out than by using a straightforward approach. For example, to evaluate m^{19} where m is an arbitrary integer, we can compute

$$m \times m \times m \times \bullet\bullet\bullet \times m$$

which requires 18 multiplications, or the multiplications can be grouped as

$$(((m \times m)^2)^2 \times m)^2 \times m$$

which requires only 6 multiplications.

To perform the grouping automatically the exponent e is treated as a program that causes squaring or multiplication to be carried out in the correct sequence. The exponent is considered a bit string; in our example, 19_{10} is 10011_2. The bit string is read from left to right (right to left can also be made to work) and if the bit read is '0' then only

squaring takes place; otherwise squaring is followed by multiplication by m.

To illustrate, using 10011, as the program and with the variable c initially 1, the following sequence is obtained:

1.	1st bit = 1	a) $c := c^2 = 1$
		b) $c := mc = m$
2.	2nd bit = 0	$c := c^2 = m^2$
3.	3rd bit = 0	$c := c^2 = m^4$
4.	4th bit = 1	a) $c := c^2 = m^8$
		b) $c := mc = m^9$
5.	5th bit = 1	a) $c := c^2 = m^{18}$
		b) $c := mc = m^{19}$

4.3 Approaches that are Infeasible in Practice

This section is a discussion of some ideas for improving the speed of modular exponentiation that are inapplicable for use with the RSA cryptosystem. In our discussion 512-bit numbers are used as a basis for illustration, for three reasons:

(1) 512 bits is approximately equivalent to 155 decimal digits, which provides acceptable but somewhat less security than the 200-digit key recommended by Rivest, et al. Note that factorization of a 512-bit key would still take many thousands of years. An SCN using keys of 512 bits would be acceptably secure.

(2) Some multiplication algorithms require that the multiplicands have lengths that are powers of 2. If the lengths are not powers of 2 the multiplicands are

padded with 0's. For clarity it is preferable to avoid discussion of the padding process and other trivial details.

(3) A consistent basis for comparison of algorithms is needed.

The infeasible approaches discussed below are

(1) Reducing the Number of Multiplications and Squarings,

(2) The Willoner Parallel Multiplier,

(3) The Toom-Cook Algorithm,

(4) The Chinese Remainder Algorithm, and

(5) The Schönhage-Strassen Algorithm.

(1) Reducing the Number of Multiplications and Squarings

The method of repeated squaring and multiplication does not always provide the optimal grouping of squarings and multiplications. Knuth[29, 'Evaluation of Powers'] analyses the problem of finding an optimal grouping. He discusses four related concepts but only one of these can be applied to our problem. This concept is that of forming an addition chain in which the shortest possible sequence of additions is generated that having as its sum the exponent to be used; the sequence of additions provides the program for squaring and multiplication. Unfortunately, finding a minimal

addition chain for an arbitrary large integer is extremely difficult, so this idea is unuseable without further research.

(2) The Willoner Parallel Multiplier

Willoner[51] designed an O(n) parallel multiplier which he suggests might find application in cryptography. If built, it would certainly make encryption much faster. No parallel divider would be required if the PFRA algorithm developed later in this thesis were used. Unless there is an unanticipated demand for the device, however, it is likely to be expensive for the foreseeable future.

(3) The Toom-Cook Algorithm

Knuth[29] discusses the Toom-Cook algorithm in detail; it is a recursive, divide-and-conquer, generalization of the Karatsuba algorithm. Unlike the Karatsuba algorithm, which splits multiplicands each into two halves and uses a special multiplication order to multiply halves, the Toom-Cook algorithm splits multiplicands into r+1 portions with r increasing with the lengths of the multiplicands. That is, the longer the multiplicands the more parts they are broken into.

Study of the algorithm shows that it proceeds in 'steps', in that it requires multiplicands of 32 bits or 80 bits or 320 bits or 1280 bits, etc. to be placed on a stack initially. In other words, multiplicands must be padded to

these lengths, and only these, before the algorithm proceeds. If the multiplicands were 320 bits in length the multiplication might proceed efficiently, but multiplicands of 512 bits require padding to 1280 bits. This amount of padding would cause a great deal of extra work to be done, so much so that we estimate that the algorithm would take 4 or 5 times longer than standard multiplication to multiply 512-bit numbers, without considering other overhead. The algorithm is therefore inapplicable without more study.

(4) The Chinese Remainder Algorithm

The Chinese Remainder Theorem provides a technique for converting from residue representation to the standard radix representation. To convert from radix to residual representation a number is divided by some relatively prime numbers (moduli) and the remainder, or residue, of each division is retained. Numbers in residue form can be added, subtracted, or multiplied in O(n) time, but no efficient way of carrying out division is known. After the desired operations are done the result is converted back to radix form.

Aho, Hopcroft, and Ullman[7] show that the time complexity of converting between radix notation and residue notation is $O(M(bk)\log_2 k)$ where b is the number of bits in each of the relatively prime moduli, k is the number of moduli, and M is the time to multiply two integers. Because the process of conversion is so time-consuming, a single

multiplication of two numbers cannot be done efficiently; gains in speed are realized only when the CRA is used to do a large number of consecutive multiplications.

Modular exponentiation requires that a remainder be obtained after each multiplication or squaring. If this requirement is strictly adhered to, use of the CRA does not make sense because frequent conversion between notations will be necessary to permit remainders to be taken.

There is actually no reason why a remainder *must* be obtained after each multiplication because, as previously noted, the remainder is taken to keep the numbers "in a manageable range". The range might be redefined so that taking the remainder is deferred until a *critical size* is reached. At that time the result would be converted into standard notation, *collapsed* by taking the remainder, converted back into residual form, and multiplication and squaring could proceed as before.

Unfortunately, the critical size/collapse approach has several deficiencies. Representing numbers up to the critical size requires either extra prime moduli or larger prime moduli. Consequently, the work done in conversion between residue and standard representation increases. This increased work factor cancels any gains obtained by postponement of division to obtain the remainder.

Also, the time required to obtain the remainder when the critical size is reached is also significant. If the critical size is set to a large value to defer several divisions, then the time required to carry out division of a number at the critical size will not be inconsequential, even using a fast division algorithm.

It appears that more research is required before use of the CRA becomes practical in this application.

(5) The Schönhage-Strassen Algorithm

This is a recursive divide-and-conquer algorithm requiring that the multiplicands be represented by their Fourier Transform. Once the multiplicands are transformed, pairs of elements of the vectors obtained by transformation are multiplied together to form a result vector. The resulting vector is transformed to radix notation using the Inverse Fourier Tranform and some further manipulation.

The Schönhage-Strassen Algorithm is asymptotically faster than any other multiplication algorithm, but it appears to be unsuitable for the multiplication of 512-bit numbers. The algorithm is clearly described in Aho, Hopcroft, and Ullman[7]; their notation is used throughout the following discussion.

To obtain a 1024-bit result the algorithm requires that the 512-bit multiplicands be padded with 0's to make them 1024-bit numbers before they are transformed using the FFT. No problems arise during the processes of transformation,

multiplication and inverse transformation until step 4c) in Algorithm 7.3 is reached. At this point the numbers $\hat{u}$ and $\hat{v}$ must be multiplied and each is 480 bits long, which is obviously not a significant improvement in comparison with the original 512-bit numbers.

The problem lies in the derivation of $\hat{u}$ and $\hat{v}$, which is done in step 4b). $\hat{u}$ and $\hat{v}$ are derived by stringing together portions of numbers called u' and v', along with some intervening 0's. u' and v' are each broken into $b = 32$ portions, each portion being $\log_2 b = 5$ bits in length. The portions when concatenated have intervening gaps of 0's, each gap $2\log_2 b = 10$ bits in length. Therefore, $\hat{u}$ and $\hat{v}$ each become $3b\log_2 b = 3(32)(5) = 480$ bits long.

The algorithm is effective for very large multiplicands because $3b\log_2 b$ for large numbers is comparatively small. For instance, with multiplicands of 64K bits, $3b\log_2 b = 6144$ so that $\hat{u}$ and $\hat{v}$ are each less than 1/10 of the length of the initial multiplicands. For 512-bit multiplicands this reduction factor is still very small so the algorithm is inefficient.

Research is required before the Schönhage-Strassen algorithm will become useable in this application. Perhaps a method might be devised to obtain the remainder without converting into radix notation.

4.4 Approaches that Improve Timing

4.4.1 Multiplication using the Karatsuba Algorithm

The Karatsuba Algorithm (see [1,3,4,7]), first published in 1962, uses a simple divide-and-conquer strategy. Two integer multiplicands a and b, each consisting of n digits (radix r), are written as

$$a = (r\uparrow n/2)a_1 + a_0 \quad \text{and,}$$

$$b = (r\uparrow n/2)b_1 + b_0.$$

The product c=ab is then computed using the *Karatsuba equation*

$$c = (r\uparrow n + r\uparrow n/2)a_1b_1 + (r\uparrow n/2+1)a_0b_0 + (r\uparrow n/2)(a_0-a_1)(b_1-b_0)$$

which requires 3 multiplications of n/2-digit numbers, along with some adding, subtracting, and shifting. Using the standard multiplication technique 4 multiplications of n/2-digit numbers would have to be done, so the Karatsuba technique saves approximately 1/4 of the work ordinarily done. The equation can be reapplied recursively to each of its 3 products, so if n is a power of 2 then $(3/4)\uparrow\log_2 n$ of the single-digit multiplications are done as would be done using the standard algorithm, if recursion is carried to the point where only single digits are being multiplied. The time complexity of the algorithm is $O(n\uparrow\log_2 3)$ or approximately $O(n\uparrow 1.59)$.

Since use of recursion adds considerable overhead, Moenck[4] suggests that standard multiplication be used when the numbers to be multiplied are 8 digits long. The following algorithm for recursive Karatsuba multiplication uses standard multiplication when the multiplicands are 'minsize' digits long:

```
1.  proc recursive karatsuba = (ref()int a,b,result)void:
2.  begin

3.     int n := upb a; (1:n/2)int a1,b1,a0,b0;
4.     (1:2*n)int a1b1,a0b0,third term;
5.     a1 := a(1:n/2); a0 := a(n/2+1:n);
6.     b1 := b(1:n/2); b0 := b(n/2+1:n);

7.     if n ≤ minsize then standard mult(a,b,result)
8.     else begin

9.       recursive karatsuba(a1,b1,a1b1);
10.      recursive karatsuba(a0,b0,a0b0);
11.      recursive karatsuba(a0 - a1,b1 - b0,third term);

12.      shift and add(a1b1,a0b0,third term,result)

13.    end
14.    fi

15. end;
```

Even the use of Moenck's idea for limiting recursion will provide only marginal improvement in practice. However, two ways of applying the Karatsuba technique suggest themselves that will provide significant gains in multiplication speed:

(1) In-line code and,

(2) Parallel implementation.

(1) **In-line code.**

Michelman[36] refers to the splitting of multiplicands into 8 pieces each before multiplication using the Karatsuba technique and states that multiplication then proceeds twice as fast, including overhead, as with use of standard multiplication. This idea is applicable only when the size of the multiplicands is known in advance, as in application to the RSA cryptosystem. The idea is simply to carry out recursion *manually* to generate an equation that can be used in a program. Depending on the size of the multiplicands and the machine word size, more than one subroutine may have to be written and a cascade of subroutine calls used.

Thus, on a computer with a hardware 16-bit multiply, to multiply two 512-bit numbers an equation is written to split the numbers into 32 one-digit pieces; with overhead, multiplication speed is improved by a factor of approximately 4 compared to standard multiplication because $3^5=243$ integer multiplications have to be done instead of the $32^2=1024$ multiplications done using the standard algorithm.

Alternatively, the 512-bit multiplicands can first be split into 8 pieces each, and multiplication of these pieces can take place using another routine which splits multiplicands into 4 pieces each. The first approach, using a single routine, provides somewhat faster multiplication speed than the second approach, but it requires more bytes

of code to be written and retained in memory.

(2) **Parallel Implementation.**

Consider lines 9 to 11 of the recursive Karatsuba algorithm detailed previously. On a parallel machine the following code can be used to replace these three lines:

```
9.      par begin
10.       recursive karatsuba(a1,b1,a1b1),
11.       recursive karatsuba(a0,b0,a0b0),
12.       recursive karatsuba(a0 - a1,b1 - b0,third term)
13.     end;
```

Note that the computation done in each of the lines 10, 11, and 12, is independent of the computation done in any other line, and that all preliminary computations are completed in lines 1 to 8. Clearly, $3\uparrow t$ processors can be used to do the computations in lines 9 to 13, where t is the depth of recursion desired.

It should be possible to build a parallel Karatsuba multiplier on a chip. The time complexity of such an implementation is $O((n\uparrow\log_2 3)/(3\uparrow t))$ which approximates O(n) for small n or large t or both. The device may be easier to construct than the Willoner Parallel Multiplier.

4.4.2 The Preconditioned Fast Remainder Algorithm

a. The Concept

From basic modular arithmetic, if h = g×i + j×k, h **mod** p can be obtained by computing

h **mod** p = (g×(i **mod** p) + j×(k **mod** p)) **mod** p.

That is, **mod**'s can be taken at any time.

The algorithm of this section, the Preconditioned Fast Remainder Algorithm or PFRA, exploits this fact. The dividend A, which is a digits long, is rewritten as

$$A_1(r\uparrow(a-z)) + A_0.$$

z is the *interval selector*. A number of iterations is required to compute A **mod** n and z is set to a different value for each iteration. During each iteration $r\uparrow(a-z)$ **mod** n is looked up in a precomputed table, multiplied by A_1 using a fast multiplication algorithm, and added to A_0 to form a result congruent to A **mod** n. Iteration terminates when the result has the same number of digits as n.

b. Development

Let A and n be integers of length a and m digits respectively. We want to find A **mod** n. Let a be greater than m, and A(p:q) represent the p'th through q'th digits of A. A may be represented as the polynomial

$$A(1:z)(r\uparrow(a-z)) + A(z+1:2z)(r\uparrow(a-2z)) +\ldots+$$
$$A(m-2z+1:m-z)(r\uparrow(m+z)) + A(m-z+1:m)(r\uparrow m) + A_{\circ}$$

where z is the *interval selector*, r is the radix, and $A_{\circ}$ represents the m low-order digits of A.

A **mod** n can be obtained as follows:

(1) If $x \geq m$ find the modulus of all the terms of the form $r\uparrow x$. That is, find $(r\uparrow(a-z))$ **mod** n, $(r\uparrow(a-2z))$ **mod** n, ..., $(r\uparrow(m+z))$ **mod** n, $(r\uparrow m)$ **mod** n.

(2) Multiply each of the results obtained in step (1) by its corresponding portion of A. That is, compute $A(1:z)(r\uparrow(a-z))$ **mod** n, $A(z+1:2z)(r\uparrow(a-2z))$ **mod** n, and so forth.

(3) Add the products obtained at step (2). Add $A_{\circ}$ to the result.

(4) If the result in step (3) has more digits than n go to step (1), otherwise stop.

Eventually the result will be m digits long. At that point if the result is still larger than the modulus standard division by n can be carried out to obtain the final result; the division will take little time since the dividend and the modulus will be the same length.

If the same modulus n is used repeatedly, as is the case in encipherment of a message by modular exponentiation, the values required in (1) can be precomputed, placed in a table, and used as required. Each entry in the table will be approximately m digits in length.

For example, let A = 36302956, n = 2357, a = 8, m = 4, and z = 2.

(1) 10^4 **mod** n = 632, 10^4 **mod** n = 572.

(2) Rewrite A as $(36)\times10^4 + (30)\times10^4 + 2956$. Compute $(36)\times(632) = 22752$, $(30)\times(572) = 17160$.

(3) Compute A' = 22752 + 17160 + 2956 = 42868.

(4) 42868 has 5 digits and n has 4, so iterate.

(1) 10^4 **mod** n = 572 (already computed).

(2) Rewrite A' as $(4)\times10^4 + 2868$. Compute $(4)\times(572) = 2288$.

(3) Compute A'' = 2288 + 2868 = 5156.

(4) A'' and n have the same number of digits so stop.

Since A'' is greater than n, standard division can now be used to obtain the final answer, 442.

If A is a large number and z is chosen correctly a fast multiplication algorithm can be used to carry out the multiplications at step (2), reducing the time complexity of finding the remainder. The problem is to choose z such that at step (4) of every iteration the result has fewer digits than at step (1) and to make the choice optimal.

If z is chosen to be (a - m)/2, where a is the number of digits in A at the beginning of each iteration, a polynomial for A can be derived that is quite good and perhaps optimal.

If A is initially 2m digits, as in modular exponentiation, another way of expressing z is as z = m/(2↑i), at the ith iteration. With this choice of z, A will always be split into 3 parts at step (2) of every iteration, as follows:

$$A(1:z)(r\uparrow(a-z)) + A(z+1:2z)(r\uparrow m) + A_0.$$

Note that in the second term replacement of '(r↑m)' by a table entry is simply replacement of an m-digit number by another m-digit number and is of no value. Therefore, we recommend that A be split into just 2 parts at the beginning of each iteration:

$$A(1:z)(r\uparrow(a-z)) + A_0.$$

This split requires just <u>one</u> table entry for each iteration.

Now, consider what happens to a 2m-digit A if z is chosen as above. On the first iteration z = m/2, so A is split as:

$$A(1:m/2)(r\uparrow(3m/2)) + A_0.$$

After multiplication and addition we get a 3m/2-digit result. On the second iteration A is split as:

$$A(1:m/4)(r\uparrow(5m/4)) + A_0$$

to obtain a 5m/4-digit result at step (4). In other words, at step (4) of every iteration we have an $m + (m/(2\uparrow i))$-digit result. Since $m/(2\uparrow i)$ becomes small rapidly, only a few iterations are required to collapse a 2m-digit number; in fact, the number of iterations, and entries in the table, is $\log_2 m$.

Let P be some multiplication algorithm which takes time M(m,m) to multiply m-digit numbers. Let k be the ratio M(m,m)/M(m/2,m/2).[1] Let R(2m,m) be the time taken to find the remainder of a 2m-digit number when divided by an m-digit number using the PFRA with a multiplication algorithm P used at step (2). Then,

$$R(2m,m) = M(m,m)\{2/k + 4/k^2 + 8/k^3 + \ldots\}$$
$$\leq M(m,m) + \epsilon \quad \text{(if P = standard algorithm; } k \leq 4\text{)}$$
$$\leq 2M(m,m) + \epsilon \quad \text{(if P = Karatsuba algorithm; } k \leq 3\text{)}$$

ϵ goes to zero asymptotically, so in this application it will be small since m is large. Therefore, in application to modular exponentiation, the PFRA will find remainders in less than twice the time to multiply or square c, if the Karatsuba algorithm is used to multiply at step (2).

---

[1] Actually k is not constant for a particular algorithm P, but rises asymptotically to a limit. Thus, k=4 in the limit for the standard multiplication algorithm but is actually 3.96 if m=80 is substituted in the timing formula for multiplication given in Chapter Three (footnote, Section 3.4.1).

c. The Algorithm

Algorithm 4.1 is the Preconditioned Fast Remainder Algorithm; we provide a small example to illustrate its operation.

Assume that 'a' is 16 digits in length and the modulus n is 8 digits long. Assume a precomputed table of terms of the form (r↑x) **mod** n with each table entry being 8 digits long. Further assume that 'a' is to be collapsed until it is 8 digits in length; that is, the variable 'limit' is set to 8.

(1) 'pfra' is entered with 'a' and 'limit'. Since 16 > 8 the loop is entered.

(2) 'collapse' is called with 'a' and a pointer to the first table entry. 'z' is computed as **even**((16-8)/2) = 4. (**even** is a function that returns the nearest even number to a given real; this is done for convenience since some fast multiplication algorithms, such as the Karatsuba algorithm, require multiplicands with an even number of digits.)

(3) 'collapse' is entered. The low-order 12 digits of 'a' are placed in 'a0' temporarily (i.e., a0 = a(5:16)). 'mhote' is called with 'a(1:4)' and the first table entry as parameters.

```
co
   'ms' or 'multiply and shift' multiplies two integers
   a and b, both the same length, using 'fast mult'.
   The result is shifted left by 'shift' digits and
   returned in the result vector 'r', which must be
   long enough. 'zero' is a routine that zeroes out a
   vector.
                                                    co

1. proc ms = (ref()int a,b,r,ref int shift)void:
2. begin

3.   zero(r);
4.   r(upb r-shift-2*upb a:upb r-shift) := fastmult(a,b)

5. end;
```

Algorithm 4.1a. 'ms'

```
co
   'mhote' or 'multiply high order and table entry'
   multiplies a z-digit number ('ho') with a table
   entry indexed by 'te'. The table is assumed globally
   available. 'get te group' obtains the ith group of z
   digits from the table entry. Table entries must be
   multiples of z digits in length.
                                                    co

 6. proc mhote = (ref()int ho,r,int te)void:
 7. begin
 8.   int z := upb ho, shift := -1;
 9.   (1:upb r)int r1; zero(r); zero(r1);
10.   (1:z)int te group;

11.   for i from (upb table(te))/z by -1 to 1
12.   do

13.     get te group(te group,table(te),i,z);
14.     ms(te group,ho,r1,shift +:= 1);
15.     add(r,r1,r)

16.   od

17. end;
```

Algorithm 4.1b. 'mhote'

```
co
   'collapse' extracts the high-order 'z' digits of 'a',
   the number to be collapsed, and multiplies them with
   a table entry indexed by 'te'. It adds the result to
   'a0', the low-order digits of a, and returns the
   final collapsed result in 'a'.
                                                      co
18. proc collapse = (ref()int a,ref int z,te)void:
19. begin
20.   (1:upb a)int a0, r; zero(a0); zero(r);
21.   a0(z+1:upb a) := a(z+1:upb a);

22.   mhote(a(1:z),r,te);
23.   zero(a);
24.   add(a0,r,a)

25. end;
```

Algorithm 4.1c. 'collapse'

```
co
   'pfra' or the 'preconditioned fast remainder
   algorithm' collapses the number 'a' until it is
   'limit' digits in length. 'get z' computes the
   interval selector 'z' by computing
   EVEN((length(a)-m)/2), where EVEN returns the nearest
   even number to a given real. 'length' is a function
   that determines the number of significant digits in
   'a' (i.e., not counting high-order zeroes).
   'te' indexes into the precomputed table.
                                                      co

26. proc pfra = (ref()int a,limit)void:
27. begin
28.   int z, te := 0, m = upb a;

29.   while length(a) > limit
30.   do
31.     collapse(a,z := get z(a,m), te +:= 1)
32.   od

33. end;
```

Algorithm 4.1d. 'pfra'

(4) 'mhote' is entered. 'a(1:4)' is multiplied by the table entry, requiring the loop to be iterated 4 times. 'ms' is used to carry out multiplications, with shifting as necessary. 'mhote' returns with a 12- or 13-digit result, 'r'.

(5) 'collapse' adds 'r' to 'a0', yielding a 12- or 13-digit result which is placed in 'a'.

(6) Since 'a' is still longer than 8 digits, steps (1) to (5) above are repeated with the new 'a'.

Eventually the algorithm terminates with 'a' 8 digits long.

4.5 Conclusion

Using the improved algorithms developed, modular exponentiation can be done in the following way on a 16-bit microprocessor. Assume that the routines 'K32 digits', 'K16 digits', 'K8 digits', 'K4 digits', and 'K2 digits' are available. These routines multiply two numbers by first breaking them into 32, 16, 8, 4, or 2 pieces respectively, depending on the lengths of the multiplicands. Assume a key of 512 bits.

(1) The routine 'K32 digits' is used for multiplication. The result of each multiplication is a 1024-bit, or 64-digit (radix 2^{16}), number. Recall that this requires 243 integer multiplications to be done.

(2) To obtain the remainder after multiplication the PFRA calls 'K16 digits' twice, 'K8 digits' 4 times, 'K4 digits' 8 times, and 'K2 digits' 16 times. The integer that is left after these calls is 34 digits long. A standard division is used to reduce the result to 32 digits, or 512 bits. Counting only the single-digit multiplications that must be done to obtain the remainder and not considering overhead, we see that standard division takes $32^2 = 1024$ integer multiplications, whereas the PFRA will use approximately 450 if used in conjunction with standard division when 'a' gets small.

Therefore, the total work to do a multiplication and to obtain the remainder is 243+450 or approximately 700 integer multiplications, which is about one-third of the number that must be done using the standard algorithms. In other words, a factor of improvement of slightly less than 2 over Michelman's implementation can be expected (if overhead is included), justifying our earlier assumptions in Chapter Three.

Finally, recall (from Chapter Three) that the use of modular exponentiation for prime generation differs from its use for encryption in that the modulus changes frequently. Since the PFRA requires that a table be constructed and used for some time it would appear that prime number generation cannot be done using the PFRA. It is possible, however, to

use a *dynamic table* for the purpose of prime generation in which the entries are decremented as the modulus is incremented. This should be easily implemented.

Summary and Conclusions

It has been observed on numerous occasions that access to information will gain importance in the society of the future; if this is true then the converse is also true: that *non-access* to information by *unauthorized* persons will become increasingly important. Therefore, it may be expected that with the anticipated rapid increase in the use of personal computers in the years to come that a secure personal communications network will find wide application. People will vote, conduct business, and send unforgeable mail electronically in the privacy of their home or office.

This thesis has shown that it is now possible to build an inexpensive, microprocessor-based, public-key secure communications and authentication network. A protocol has been outlined wherein network users generate their own keypairs and, because of this distributed key generation, they are protected against forgeries and active use of a stolen or lost secret key. It has been shown that the channel capacity and key generation speed of such a network may be acceptable for some interpersonal applications if the latest microprocessors are used, incorporating algorithms described in the thesis.

The actual implementation of the proposed network will take some time and effort. Although some of the necessary software has been implemented in a high-level language, there is much work left to do in rewriting RSACRYPT into the

assembly language of the microcomputer to be used, implementation of the algorithms described in Chapter Four, implementation of the software needed by the key distribution center, and design and implementation of software to allow mail facilities.

If it should be decided that the proposed network be built we recommend that work proceed, in parallel with software implementation, on the design of an improved serial algorithm for even faster modular exponentation. We expect that the algorithms described in this thesis for multiplication and finding the remainder of a division are not the last word in what could be done: it is not far-fetched to believe that a truly real-time microprocessor-based network might be built, in which people would communicate at 60 words per minute or more.

Another interesting topic for further research is the construction of a parallel Karatsuba multiplier. Such a device would have application in areas outside cryptography, particularly if it were inexpensive. It might be designed to permit the addition of components at any time, thus allowing the user to set multiplication speed at the level desired.

References

[1] Aho, A.V., J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974

[2] Beker, H.J., "Security in an Electronic Fund Transfer System", *Information Privacy*, vol. 2, no. 5, Sept. 1980 (8 Refs)

[3] Bright, H.S. and R.L. Enison, "Quasi-Random Number Sequences from a Long-Period TLP Generator with Remarks on Applications to Cryptography", *Computing Surveys*, Vol. 11, No. 4, Dec. 1979 (41 Refs)

[4] Bright, H.S. and R.L. Enison, "Cryptography Using Modular Software Elements", *Proc. AFIPS NCC*, Vol. 45, AFIPS Press, Arlington, Va., pp113-123, 1976

[5] Cabay, S., *Personal Communication*, University of Alberta, May 1981

[6] Chaum, D.L., "Untraceable Electronic Mail, Return Addresses and Digital Pseudonyms", *Communications of the ACM*, Vol. 24, No. 2, Feb. 1981 (5 Refs)

[7] Davies, D.W. and D.A. Bell, "Protection of Data by Cryptography", *Information Privacy*, Vol. 2, No. 3, May 1980 (3 Refs)

[8] Davies, D.W., W.L. Price and G.I. Parkin, "Evaluation of Public-Key Cryptosystems", *Information Privacy*, Vol. 2, No. 4, July 1980 (17 Refs)

[9] Denning, D., "Secure Personal Computing in an Insecure Network", *Communications of the ACM*, Vol. 20, No. 7, July 1977 (31 Refs)

[10] Denning, D. and P. Denning, "Certification of Programs for Secure Information Flow", *Communications of the ACM*, Vol. 20, No. 7, July 1977 (31 Refs)

[11] Denning, D. and P. Denning, "Data Security", *Computing Surveys*, Vol. 11, No. 3, Sept. 1979 (101 Refs)

[12] Diffie, W. and M.E. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, Vol. IT-22, No. 6, Nov. 1976 (14 Refs)

[13] Doler, D. and A.C. Yao, "On the Security of Public-Key Protocols", *Stanford University*, STAN-CS-81-854, May 1981 (Abstract).

[14] Fabry, R.S., "Capability-Based Addressing", *Communications of the ACM*, Vol. 17, No. 7, July 1974 (18 Refs)

[15] Feistel, H., "Cryptography and Computer Privacy", *Scientific American*, Vol. 228, No.5, May 1973

[16] Fenton, J.S., "Memoryless Subsystems", *The Computer Journal*, Vol. 17, No. 2, Jan. 1973 (3 Refs)

[17] Gardner, M., "A New Kind of Cipher That Would Take Millions of Years to Break", *Scientific American*, Vol. 236, No. 8, Aug. 1978

[18] Gordon, J., "Use of Intractable Problems in Cryptography", *Information Privacy*, Vol. 2, No. 5, Sept. 1980 (10 Refs)

[19] Graham, G.S. and P. Denning, "Protection - Principles and Practice", *Proc. 1972 AFIPS Spring Jt. Computer Conf.*, Vol. 40, AFIPS Press, Montvale, N.J. (20 Refs)

[20] Greguras, F., "Corporate EFT: Vulnerabilities and Other Audit Considerations", *Information Privacy*, Vol. 3, No. 3, May 1981 (0 Refs)

[21] Hellman, M.E., "An Extension of the Shannon Theory Approach to Cryptography", *IEEE Transactions on Information Theory*, Vol. IT-23, No.3, May 1977 (15 Refs)

[22] Hellman, M.E., "Security in Communication Networks", *National Computer Conference*, 1978 (18 Refs)

[23] Hellman, M.E., "The Mathematics of Public-Key Cryptography", *Scientific American*, Vol. 241, No. 2, Aug. 1979 (4 Refs)

[24] Hindin, H.J., "LSI-Based Encryption Discourages the Data Thief", *Electronics*, June 21, 1979

[25] Horowitz, E., "Modular Arithmetic and Finite Field Theory: A Tutorial", *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, Mar. 1971 (22 Refs)

[26] Kahn, D., *The Code Breakers: The Story of Secret Writing*, MacMillan, New York, 1967

[27] Kam, J.B. and G.I. Davida, "Structured Design of Substitution-Permutation Encryption Networks", *IEEE Transactions on Computers*, Vol. c-28, No. 10, Oct. 1979

[28] Kline, C.S. and G.J. Popek, "Public-Key vs. Conventional-Key Encryption", *AFIPS National Computer Conference Proceedings*, New York, N.Y., Vol. 48, June 4-7, 1979 (13 Refs)

[29] Knuth, D.E., *The Art of Computer Programming, Vol. 2: Semi-Numerical Algorithms*, Addison-Wesley, 1971

[30] Lampson, B.W., "A Note on the Confinement Problem", *Communications of the ACM*, Vol. 16, No. 10, Oct. 1973

[31] Lempel, A., "Cryptology in Transition", *Computing Surveys*, Vol.11, No.4, Dec. 1979 (47 Refs)

[32] Lennon, R.E., "Cryptography Architecture for Information Security", *IBM Systems Journal*, Vol. 17, No. 2, 1978 (9 Refs)

[33] Levinson, S.E. and M.Y. Liberman, "Speech Recognition by Computer", *Scientific American*, Vol. 244, No. 4, April 1981 (2 Refs)

[34] Merkle, R.C., "Secure Communications Over Insecure Channels", *Communications of the ACM*, Vol. 21, No. 4, April 1978 (7 Refs)

[35] Merkle, R.C. and M.E. Hellman, "Hiding Information and Signatures in Trapdoor Knapsacks", *IEEE Trans. Inf. Theory*, Vol. IT-24, No. 5, Sept. 1978 (14 Refs)

[36] Michelman, E.H., "The Design and Operation of Public-Key Cryptosystems", *AFIPS NCC Proceedings*, New York, N.Y., Vol. 48, June 4-7, 1979 (8 Refs)

[37] Moenck, R.T., "Practical Fast Polynomial Multiplication", *Proceedings of the 1976 Symposium on Symbolic and Algebraic Computation*, 1976 (20 Refs)

[38] Nelson, J., "The Development of Commercial Cryptosystem Standards", *Cryptologia*, Vol. 4, No. 4, Oct. 1980 (5 Refs)

[39] Needham, R. and M. Schroeder, "Security and Authentication in Large Networks of Computers", *Communications of the ACM*, Vol.21, No.12, Dec. 1978

[40] Popek, G. and C. Kline, "Encryption and Secure Computer Networks", *Computing Surveys*, Vol. 11, No. 4, Dec. 1979 (48 Refs)

[41] Rivest, R.L., "Remarks on a Proposed Cryptanalytic

Attack on the MIT Public-Key Cryptosystem", *Cryptologia*, Vol. 2, No. 1, Jan. 1978

[42] Rivest, R.L., A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *MIT Technical Report*, MIT/LCS/Tm-82, April 1977 (11 Refs)

[43] Shamir, A., "How to Share a Secret", *Communications of the ACM*, Vol. 22, No. 11, Nov. 1979 (5 Refs)

[44] Shankar, K.S., "The Total Computer Security Problem: An Overview", *COMPUTER*, June 1977

[45] Shannon, C.E., "Communication Theory of Secrecy Systems", *Bell System Technical Journal*, Vol. 28, Oct. 1949, 656-715

[46] Simmons, G.J., "Symmetric and Asymmetric Encryption", *Computing Surveys*, Vol. 11, No. 4., Dec. 1979 (67 Refs)

[47] Simmons, G.J. and M.J. Norris, "Preliminary Comments on the MIT Public-Key Cryptosystem", *Cryptologia*, Vol. 1, No. 4, Oct. 1977, 406-414

[48] Sugarman, R., "On Foiling Computer Crime", *IEEE Spectrum*, July 1979

[49] Whiteside, T., "Annals of Crime: Dead Souls in the Computer", *New Yorker*, Aug. 22, 1977

[50] Williams, D. and H.J. Hindin, "Can Software Do Encryption Job?", *Electronics*, July 3, 1980

[51] Willoner, R.G., "On the Design of a Parallel Arithmetic Unit", *Ph.D. Thesis*, University of Alberta, Fall 1980 (59 Refs)

Appendix 1 - An Example of DES

In what follows we show how DES (the 'Data Encryption Standard') encrypts and decrypts information, to provide some intuitive understanding of the algorithm. Obviously, since DES works with blocks of 8 characters or 64 bits at a time and puts each block through 16 rounds of bit-shuffling, it would be tedious to work an example of the full operation of the algorithm. Therefore, this example will show the operation of a condensed version of the algorithm by working with a reduced alphabet, by defining a byte as being only 4 bits long, and by putting a block through only 2 rounds of bit shuffling.

A.1 Preliminaries

The alphabet that is used has only 16 characters, each character therefore needing only 4 bits to represent it. It is a meaningless alphabet as defined by the following string: '0123EYABCDUSHKRT'. The bit representations for the characters in the string range consecutively from 0000_2 for '0' to 1111_2 for 'T'.

As it happens, it is possible to spell at least two words with the above alphabet: 'DATAKEY' and 'SACHARUK'. 'DATAKEY' consists of 7 characters and therefore 28 bits and will be used as the initial encryption key in the example to follow. 'SACHARUK' consists of 8 characters or 32 bits and is the plaintext message that will be encrypted and

decrypted. Since the full version of DES uses a 56-bit initial key and works with 64-bit message blocks, it is apparent that the key and message in our example are exactly half as long as in the full version; however, the key and message consist of the same number of characters as in the full version.

The bit representation for 'DATAKEY' is therefore

1001 0110 1111 0110 1101 0100 0101

and the bit representation for 'SACHARUK' is

1011 0110 1000 1100 0110 1110 1010 1101.

DES uses two procedures for permutation that transpose characters in a block of 8 characters that is passed as a parameter. These are IP (Initial Permutation) and IP^{-1} (Initial Permutation Inverse), defined as follows:

x<84725613> := IP(x<12345678>)

x<74825631> := IP^{-1}(x<12345678>)

where 'x<12345678>' represents the initial block of 8 bytes to be permuted, and 'x<84725613>' and 'x<74825631>' represent the results of the permutations. The permutations above are arbitrarily defined and are not necessarily the permutations used in an actual implementation of DES. A block permuted by IP is restored to its original order when permuted by IP^{-1}.

A third permutation procedure called P-BOX shuffles 8 half-bytes. In this example, P-BOX is passed 16 bits; it shuffles 8 groups of 2 bits each. We define P-BOX arbitrarily as:

x<56324817> := P-BOX(x<12345678>).

DES also uses two procedures for *permuted choice* in which a group of characters is first permuted and then one of the characters in the result is dropped or replaced by the null character. We define the procedures as:

x<4823675> := PC1(x<12345678>)

x<672135> := PC2(x<1234567>)

That is, 8 characters are passed to PC1, they are shuffled, and the character designated as '1' is dropped, leaving 7 characters. PC2 is passed a block of 7 characters that are shuffled; the byte designated as '4' is dropped leaving 6 characters in the result.

DES uses some tables, the *e-table* and *s-tables*, for *non-linear substitution* in which groups of bits are replaced by fewer, or more, bits obtained from a table. (Part of the reason for reducing the number of bits in a character in this example is to reduce the sizes of the tables, which are fairly large in an actual implementation.) We define the e-table, arbitrarily, in Figure A.1.

Input	Output
00	000
01	011
10	100
11	111

Figure A.1 - The e-table

Given '00' as an index into the e-table, '000' is returned. Thus, the e-table is actually an expansion table that

replaces 2 bits by 3. In a full implementation, 4 bits are replaced by 6, making the table 4 times larger.

We define the s-tables, arbitrarily, in Figure A.2.

S1	1	0	S2	1	0	S3	1	0	S4	1	0
00	11	10	00	01	11	00	11	00	00	10	00
01	00	10	01	10	11	01	10	10	01	00	01
10	01	11	10	00	01	10	01	00	10	10	11
11	00	01	11	00	10	11	01	11	11	11	01
S5	**1**	**0**	**S6**	**1**	**0**	**S7**	**1**	**0**	**S8**	**1**	**0**
00	11	11	00	10	11	00	01	11	00	11	01
01	01	00	01	00	00	01	11	00	01	01	10
10	10	01	10	01	10	10	10	10	10	10	00
11	10	00	11	11	01	11	01	00	11	11	00

Figure A.2 - The s-tables

The s-tables are indexed with three bits by using the first two bits to determine the row and the third bit to choose the column in a table. Therefore, 2 bits replace 3. In a full implementation, 4 bits replace 6; therefore each table in a real implementation is 8 times larger than the ones above.

A.2 Key Series Generation

Before encryption is done a series of keys is generated from the initial key, 'DATAKEY'. Key series generation is done by a procedure called KS that, in a full implementation, generates 16 keys, each 48 bits long, from the initial 56-bit key. In this example only 2 keys are generated, each 24 bits long, from the initial 28-bit key.

First, 'DATAKEY' in bits form, or

1001 0110 1111 0110 1101 0100 0101

is expanded from 28 bits to 32 by adding 4 parity bits to each of the first 4 bytes, obtaining

1001 0011 0011 1100 1100 1101 0100 0101.

In an actual implementation parity bits for all 7 bytes are added, plus 1 more parity bit obtained by taking the parity of the 56+7=63 bit result.

The 32-bit expanded key is now passed to PC1 which drops a byte (of 4 bits) yielding a 28-bit result

1100 0101 0011 0011 1101 0100 1100

that we call the *key seed*.

The key seed is split into two 14-bit halves and placed in two registers C_0 and D_0. Therefore

1100 0101 0011 00

is placed in C_0, and

11 1101 0100 1100

is put into D_0. The two registers are now rotated left with end-around carry, yielding

1000 1010 0110 01

in C_1, and

1110 1010 0110 01

in D_1. The two registers are concatenated in a third register (C_0 and D_0 are saved for later use) and the result passed to PC2. Since 28 bits are passed, 24 bits remain after shuffling and dropping a byte:

1001 1001 1000 1010 0111 1010.

This result we call *Key No. 1*.

The contents of C_0 and D_0 are now rotated left with end-around carry one more time. The register results are concatenated and passed to PC2 yielding *Key No. 2*:

0011 0011 0100 0001 1100 0101.

Key No. 1 and Key No. 2 are now used to encipher the message block as shown in Figure A.3. We describe the process of encryption with reference to Figure A.3.

A.3 Encryption

(1) 'SACHARUK' is passed to IP yielding 'KHUAARSC'.

(2) The bit representation of 'KHUAARSC' is split into two registers, L_0 and R_0.

(3) The value in R_0 is expanded by taking two bits at a time and using them to index into the e-table. The result after expansion is 24 bits long, which is the same length as Key No. 1.

(4) Exclusive Or R_0 and Key No. 1, to form the intermediate result Q. Q should be thought of as consisting of 8 blocks, each block 3 bits long.

(5) The 8 blocks in Q are used to index into the 8 s-tables and are each replaced by their corresponding 2-bit table entry. The result is 16 bits long.

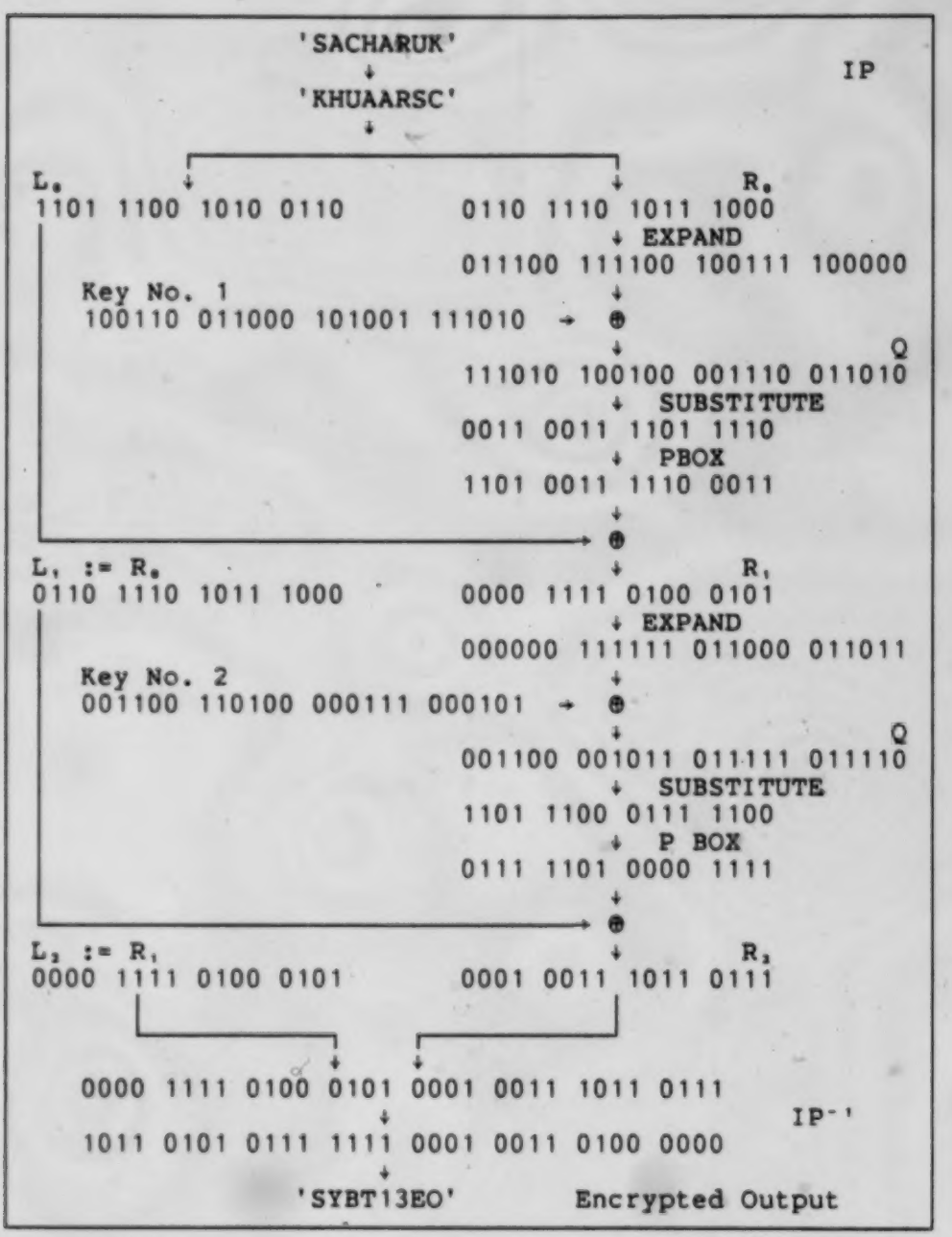

Figure A.3. Encryption with DES
(⊕ = exclusive or)

(6) The 16 bit result is sent to P-BOX which permutes 8 groups of 2 bits each.

(7) The result of P-BOX is XOR'ed with L_0 and placed in register R_1. R_0 is now placed in register L_1.

(8) The process described in steps 1 to 7 above is repeated with registers L_1 and R_1 replacing L_0 and R_0, and with Key No. 2 replacing Key No. 1.

(9) R_1 is placed in register L_2. The result at step 8 above is placed in register R_2. L_2 and R_2 are concatenated.

(10) The concatenated result at step 9 is passed to IP^{-1} yielding the encrypted output 'SYBT13E0'.

A.4 Decryption

Decryption works in a fashion extremely similar to encryption, but is not *precisely* the same. See Figure A.4.

First, 'SYBT13E0' is passed into IP yielding '0TEY13SB'. The result is then split into two halves and placed into registers L_2 and R_2. The decryption process now works with the *left* register, L_2, instead of the right register as in encryption. A second dissimilarity between encryption and decryption is that Key No. 2 is used first in decryption, followed by Key No. 1. The final dissimilarity is that in encryption the register L_2 forms the high-order bits of the

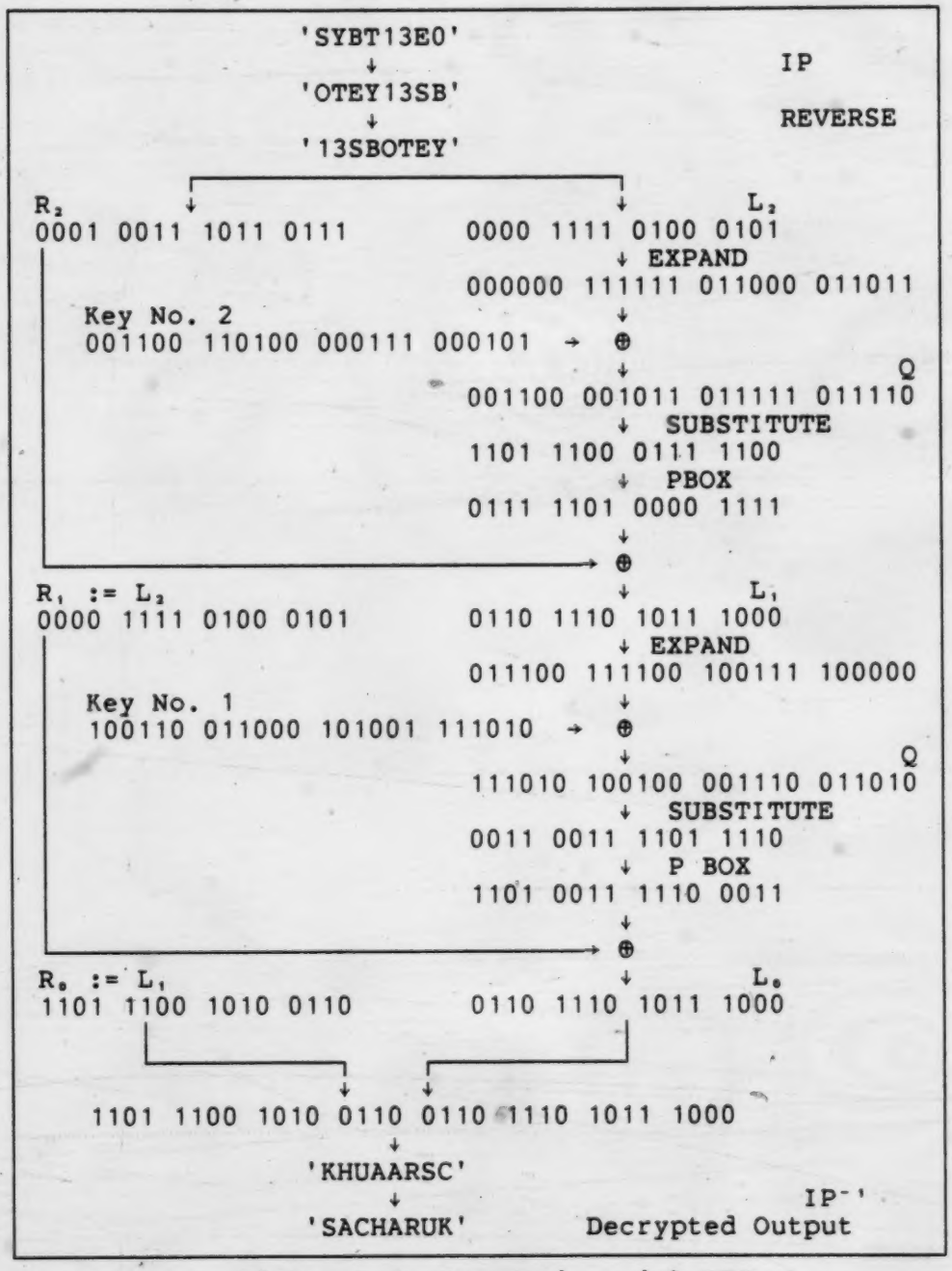

Figure A.4. Decryption with DES
(⊕ = exclusive or)

concatenated final result; in decryption R_0, the right register, has the high-order (i.e., leftmost) bits of the result.

After concatenation of R_0 and L_0 the result is passed to IP^{-1} and the original plaintext message results.